In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

University of Alberta

Efficient Algorithms in Motion Blurring and Photon Mapping

by

Daniel Neilson

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-87847-3
Our file Notre référence
ISBN: 0-612-87847-3

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canadä

University of Alberta

Library Release Form

Name of Author: Daniel Neilson

Title of Thesis: Efficient Algorithms in Motion Blurring and Photon Mapping

Degree: Master of Science

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Date: August 20, 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Efficient Algorithms in Motion Blurring and Photon Mapping** submitted by Daniel Neilson in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Yee-Hong Yang
Supervisor

Dr. Kim Chow
External Examiner

Dr. Sherif Ghali

Dr. Martin Müller

Date: Aug 18, 2005

Abstract

Ray tracing is an algorithm for synthesizing photorealistic images by tracing paths of light backwards from the camera into the scene. This thesis describes enhancements which improve efficiency and image quality.

Adaptive motion blurring increases sampling in motion blurred regions. This causes a disparity when sampling stationary objects, visible as a reduction of the colour variance. The cause and solution to this artifact are examined.

The time dependent photon map is a method proposed by Cámmarano and Jensen to model motion blurred illumination and caustics. This thesis presents a modification, the adaptive photon map, which merges time dependent and independent photon maps.

Christensen proposed an irradiance precalculation method for photon maps which results in a substantial speedup. However, it is likely that this method will precalculate values which are never used. A subtle but important change is proposed that converts this method into a cache. Significant speedups are observed.

Acknowledgements

I would like to take this opportunity to thank my fiancé Christine Marshall for her encouragement and companionship throughout the duration of my MSc program; I couldn't have done it without you.

I would also like to thank my supervisor Yee-Hong Yang for his guidance throughout the research and development process for the ideas found in this thesis. Furthermore, I would like to thank Yee-Hong Yang for his extensive help editing this thesis.

Additionally, I would like to thank Markian Hlynka for allowing me to bounce ideas off of him while I developed the ideas found in this thesis, and for dragging me away from my work to help with his whenever I needed a break.

Contents

1 Introduction 1

2 Ray Tracing 3
- 2.1 Ray-Object Intersections . . . 4
- 2.2 Light Reflection and Refraction . . . 4
  - 2.2.1 Mirror Reflecting Light . . . 5
  - 2.2.2 Refraction . . . 5
- 2.3 Direct Lighting . . . 6
  - 2.3.1 Visibility of a Light Source . . . 7
  - 2.3.2 Diffuse Surfaces . . . 7
  - 2.3.3 Specular Surfaces . . . 8
  - 2.3.4 Shading . . . 9
  - 2.3.5 Direct Lighting from Area Light Sources . . . 10
- 2.4 The Camera . . . 10
- 2.5 The Classical Ray Tracing Algorithm . . . 12
- 2.6 Global Illumination . . . 15
  - 2.6.1 Distributed Ray Tracing . . . 15
  - 2.6.2 Irradiance Caching . . . 18
  - 2.6.3 The Rendering Equation . . . 18
  - 2.6.4 Path Tracing . . . 19
  - 2.6.5 Backwards Ray Tracing . . . 19
  - 2.6.6 Photon Mapping . . . 21
- 2.7 Photon Mapping . . . 22
  - 2.7.1 Data Representation . . . 22
  - 2.7.2 Creating the Photon Map . . . 23
  - 2.7.3 Using the Photon Map to Calculate Global Illumination . . . 26
- 2.8 Motion Blurring . . . 31
  - 2.8.1 Adaptive Motion Blurring . . . 33
- 2.9 The Time Dependent Photon Map . . . 36
- 2.10 Optimizing Ray-Object Intersection Calculations . . . 38
  - 2.10.1 Bounding Volumes . . . 38
  - 2.10.2 Spatial Partitioning . . . 41
- 2.11 Efficient Shadow Calculations . . . 44
- 2.12 Precalculating Irradiance . . . 44
  - 2.12.1 Precalculating Irradiance in the Time Dependent Photon Map . . . 45

3 Efficient Algorithms 47
- 3.1 An Oversampling Problem with the Adaptive Motion Blur Algorithm . . . 48
  - 3.1.1 Variance Invariant Adaptive Motion Blurring Algorithm . . . 49
  - 3.1.2 Results . . . 50
- 3.2 On Demand Irradiance Caching with Photon Maps . . . 54
  - 3.2.1 Results . . . 55

3.3 The Adaptive Photon Map 58
3.3.1 Using On Demand Irradiance Caching 60
3.3.2 Results 60
3.4 Adaptive Photon Map with Interval Photons 65
3.4.1 Using On Demand Irradiance Caching 67
3.4.2 Results 68

4 Conclusion **73**
4.1 Contributions 73
4.2 Future Work 74

References **75**

A Glossary of Terms **77**

B Calculating the Cohen-Sutherland Outcode of a Point **78**

C Images Using Adaptive Motion Blurring **79**

D Images Using Precalculated Irradiance and On Demand Caching **81**

E Images Using the Time Dependent Photon Maps **83**

F Specifications of the Computers Used to Generate Data **85**
F.1 giroux.cs.ualberta.ca 85
F.1.1 uname -a 85
F.1.2 cat /proc/version 85
F.1.3 cat /proc/cpuinfo 85
F.1.4 cat /proc/meminfo 85
F.1.5 gcc -v 86
F.2 garner.cs.ualberta.ca 86
F.2.1 uname -a 86
F.2.2 cat /proc/version 86
F.2.3 cat /proc/cpuinfo 86
F.2.4 cat /proc/meminfo 87
F.2.5 gcc -v 87
F.3 goodfare.cs.ualberta.ca 87
F.3.1 uname -a 87
F.3.2 cat /proc/version 87
F.3.3 cat /proc/cpuinfo 87
F.3.4 cat /proc/meminfo 88
F.3.5 gcc -v 88

List of Figures

2.1 Reflection of a ray, $\boldsymbol{R}$. 5
2.2 Refraction of a ray, $\boldsymbol{R}$. 6
2.3 Geometry for diffuse lighting calculation. 8
2.4 Geometry for specular lighting calculation. 9
2.5 The pinhole camera. 11
2.6 A scene to demonstrate the classical ray tracing algorithm. 13
2.7 Direct and indirect illumination of a point $\boldsymbol{P}$. 15
2.8 Image of a caustic formed by light shinning on a metallic ring. 20
2.9 Demonstration of the adaptive motion blurring algorithm. 34

3.1 A fast moving blue sphere ray traced using the adaptive motion blurring algorithm of Section 2.8.1 with $N_s = 10$ and $N_t = 100$. 49
3.2 A fast moving sphere ray traced using the VIAMB algorithm proposed in this thesis with $N_s = 10$ and $N_t = 100$. 51
3.3 Side-by-side comparison of the images in Figure 3.1 (left) and Figure 3.2 (right). 51
3.4 Variation of the variance of the red plane both inside and outside of the motion blurred region as N_t increases. 52
3.5 Scene nine from the test suite ray traced using both adaptive motion blurring algorithms. The left image was produced using the original algorithm, and the right was produced using VIAMB. 54
3.6 Scene containing nine spheres from the moving spheres test suite. Produced using precalculated irradiance (left) and on demand caching (right) with a time dependent photon map. 55
3.7 Time (in seconds) to ray trace the odd numbered moving spheres scenes in the test suite with the time dependent photon map with both Christensen's irradiance precalculation optimization and the presented on demand caching optimization. 57
3.8 Sample image ray traced using the adaptive photon map. 60
3.9 Number of time independent and time dependent photons used when ray tracing the odd numbered moving spheres scenes in the test suite. 61
3.10 Scene containing five spheres from the moving spheres test suite. Produced using a time dependent photon map (left) and an adaptive photon map (right). 62
3.11 Time (in seconds) to ray trace the odd numbered moving spheres scenes in the test suite with the time dependent photon map and the adaptive photon map. 62
3.12 Resident memory usage (in millions of bytes) of the time dependent photon map and the adaptive photon map in the odd numbered moving spheres scenes in the test suite. 63
3.13 Peak memory usage (in millions of bytes) of the time dependent photon map and the adaptive photon map in the odd numbered moving spheres scenes in the test suite. 64
3.14 Sample image ray traced using the adaptive photon map with interval photons. 67

3.15 Scene containing five spheres from the moving spheres test suite. Produced using an adaptivee photon map (left) and an adaptive photon map with interval photons (right). 69
3.16 Number of interval, time independent, and time dependent photons used when ray tracing the odd numbered moving spheres scenes in the test suite. . 69
3.17 Time (in seconds) to ray trace the odd numbered moving spheres scenes in the test suite with the time dependent photon map, the adaptive photon map, and the adaptive photon map with interval photons. 70
3.18 Resident memory usage (in millions of bytes) of the time dependent photon map, the adaptive photon map, and the adaptive photon map with interval photons in the odd numbered moving spheres scenes in the test suite. 71
3.19 Peak memory usage (in millions of bytes) of the time dependent photon map, the adaptive photon map, and the adaptive photon map with interval photons in the odd numbered moving spheres scenes in the test suite. 71

C.1 Scene one from the test suite used to compare the quality of the two adaptive motion blurring algorithms. The left image was produced using the original algorithm, and the right was produced using VIAMB. 79
C.2 Scene 14 from the test suite used to compare the quality of the two adaptive motion blurring algorithms. The left image was produced using the original algorithm, and the right was produced using VIAMB. 80
C.3 Scene 18 from the test suite used to compare the quality of the two adaptive motion blurring algorithms. The left image was produced using the original algorithm, and the right was produced using VIAMB. 80

D.1 Scene containing three spheres from the moving spheres test suite. Produced using precalculated irradiance (left) and on demand caching (right) with a time dependent photon map. 81
D.2 Scene containing 13 spheres from the moving spheres test suite. Produced using precalculated irradiance (left) and on demand caching (right) with a time dependent photon map. 82
D.3 Scene containing 19 spheres from the moving spheres test suite. Produced using precalculated irradiance (left) and on demand caching (right) with a time dependent photon map. 82

E.1 Scene containing one sphere from the moving spheres test suite. Produced using the time dependent photon map (top), the adaptive photon map (bottom-left), and the adaptive photon map with interval photons (bottom-right). . . 83
E.2 Scenes containing seven (left) and 17 (right) spheres from the moving spheres test suite. Produced using the time dependent photon map (top), the adaptive photon map (middle), and the adaptive photon map with interval photons (bottom). 84

List of Tables

2.1 Variables and constants of Equation 2.12. 10
2.2 Child nodes to prune during kd-tree ray traversal. 43

3.1 Time (in minutes) taken to ray trace the scene used in Figure 3.4 with $N_s = 10$ on a 2.26 GHz Pentium 4 processor with 512Mb of memory running Linux kernel version 2.4.9. 52
3.2 Time (in seconds) taken to ray trace the twenty moving spheres scenes in the test suite on a 2.26 GHz Pentium 4 processor with 512Mb of memory running Linux kernel version 2.4.9. 53
3.3 Number of time dependent photons used in each of the odd numbered moving spheres scenes. 56
3.4 Comparison of pixel colours between images rendered using the time dependent photon map with Christensen's irradiance precalculation method and the high quality baseline renders. 57
3.5 Comparison of pixel colours between images rendered using the time dependent photon map with the on demand irradiance caching method and the high quality baseline renders. 57
3.6 Total number of photons used in the adaptive photon map in each of the odd numbered moving spheres scenes. 61
3.7 Comparison of pixel colours between images rendered using the adaptive photon map with the on demand irradiance caching method and the high quality baseline renders. 63
3.8 Total number of photons used in the adaptive photon map with interval photons in each of the odd numbered moving spheres scenes. 68
3.9 Comparison of pixel colours between images rendered using the adaptive photon map with interval photons plus on demand irradiance caching method and the high quality baseline renders. 70

List of Symbols

n_x	Number of horizontal pixels in an image.
n_y	Number of vertical pixels in an image.
N_s	Number of rays cast through each pixel.
N_f	Number of random secondary rays to cast during distributed ray tracing.
N_t	Number of temporal samples to use when calculating motion blur.
N_p	Number of photons to cast from light sources.
n_p	Number of photons to use when calculating an irradiance estimate.
N_{tp}	Number of time dependent photons to create when splitting a time independent photon in the adaptive time dependent photon map.
$\mathcal{V}$	The set of all lights in the scene.
$\mathcal{V}(x)$	The set of all lights visible from x.
$k_d(x)$	The diffuse colour of the material at point x. This value is also refered to as the pigment, Lambertian, or matte colour of the material at x.
$k_s(x)$	The specular, or highlight, colour of the material at point x.
$n_s(x)$	The specular exponent of the material at the point x. $n_s(x) > 0$, and controls how focused the specular highlight is.
$k_r(x)$	The mirror reflection colour of the material at point x.
$k_t(x)$	The transmission, or transparency, colour of the material at point x.
$\boldsymbol{L}_{\mathcal{L}}$	Let $\boldsymbol{P}$ be the position of light $\mathcal{L}$, then $\boldsymbol{L} = \frac{\boldsymbol{P}-\boldsymbol{x}}{\|\boldsymbol{P}-\boldsymbol{x}\|}$.
$\boldsymbol{H}_{\boldsymbol{R}}$	The mirror reflection of the ray R.
I_r	The colour of the light from the mirror reflection direction (non-zero only if the material at point x is a mirror).
I_t	The colour of the light from the transmission/refracted direction (non-zero only if the material at point x is transparent).
$I_{\mathcal{L}}$	The colour of light source $\mathcal{L}$.
$A_{\mathcal{L}}$	The surface area of light source $\mathcal{L}$; for a point light we define the surface area to be one.
$W_{\mathcal{L}}$	The wattage per unit area of light source $\mathcal{L}$.
$\Phi_{\mathcal{L}}$	The power of light source $\mathcal{L}$.

Chapter 1

Introduction

Ray tracing is a high quality photo-realistic image synthesis algorithm capable of generating images with a wide variety of lighting effects. This is accomplished by simulating the transport of light through the scene for which the image is being synthesized. An important component of high quality image synthesis algorithms is the ability to duplicate the blurring effect of moving objects in photography. In non-synthetic photography, this blurring effect arises when a component of the scene is moving at a high speed relative to the shutter speed of the camera. This thesis concerns the efficiency and accuracy of ray tracing algorithms when synthesizing scenes containing motion blur.

One efficient method of adding motion blur to the ray tracing algorithm is with a technique called adaptive motion blurring. This technique temporally samples a ray of light only when that ray passes through the path of motion of a moving object. However, the technique can visibly reduce the variance of the illumination of stationary objects in motion blurred regions of the image when compared to the same object in non-motion blurred regions. This thesis presents a new modification to the adaptive motion blurring algorithm that removes this variance discrepancy while making the algorithm more efficient.

A key element of any photo-realistic image synthesis algorithm is the ability to accurately simulate global illumination. The photon map is a data structure recently popularized by Jensen [14, 15] to enable efficient calculation of global illumination and caustics when ray tracing a scene. The structure is a record of the interactions of light with the scene that can later be queried to calculate both global illumination and caustics at a point in the scene.

When an image generation algorithm utilizes both global illumination and motion blurring, an illumination algorithm capable of calculating the global illumination of a moving object through-out its entire range of motion is required. To this end, Cammerano and Jensen [3] devise an extension of the photon mapping technique that approximates the distribution of light in time and space. In their extension, every photon is assigned a random time as it is created. The photon is then traced through the scene, at this random time, as is normally done in photon mapping. Though this method is able to produce visually stunning

images, it is inflexible relative to the characteristics of motion within the scene being ray traced. This thesis presents a new adaptive photon map that merges time dependent and time independent photons into a single photon map. This adaptive photon map increases the number of time dependent photons based on the characteristics of motion within the scene.

To further reduce the memory required by the adaptive photon map, this thesis introduces a new type of photon; the interval photon. The interval photon is a single photon that merges several time dependent photons into a single photon. The interval photon is incorporated into the adaptive photon map to test the effects of the additional photon type on memory usage, and ray tracing speed.

To increase the efficiency of utilizing a photon map to calculate the global illumination of a scene, Christensen introduced an optimization that precalculates the irradiance of the scene at each photon location in the photon map. However, this optimization may precalculate the irradiance of points that are never actually used when ray tracing the scene. To remedy this, this thesis presents an enhancement to Christensen's irradiance precalculation method into an irradiance cache. This cache only calculates the irradiance of a point if it is required when ray tracing the scene, and saves it for later use. Furthermore, the cache does not require more memory than Christensen's irradiance precalculation method.

Chapter 2 concerns algorithms and data structures required when implementing an efficient ray tracer capable of motion blur. In particular, algorithms for ray-object intersection, light refraction and reflection, direct lighting, ray tracing, global illumination, and motion blur are discussed. Four new improvements to algorithms presented in Chapter 2 are discussed in Chapter 3. In Chapter 4, the results presented by this thesis are summarized.

Chapter 2

Ray Tracing

Ray tracing is an image synthesis technique that achieves very realistic looking images by simulating light transport through the scene being rendered. Naively, we could imagine a system that simulates all rays of light originating at the light sources in the scene. Such a system would trace each ray of light through many refractions, and reflections as it interacts with the objects in the scene only to record the energy of those rays that pass into the camera to strike the film. The system would produce strikingly realistic images since it is simply our real world programmed into a computer. Unfortunately, since not all rays of light would contact the camera, an overwhelming number of calculations performed would have little or no effect on the resulting image. To overcome this obstacle, a more efficient algorithm simulates rays of light backwards along their paths. That is, it begins at the image plane, rather than at the light sources, and traces rays of light backwards into the scene until they terminate at either a light source or a diffuse surface. This approach is called ray tracing. Each ray of light that is traced into the scene contributes to the colour of the image pixel that it initially passes through. This simple idea of tracing the rays of light backwards into the scene, from the camera, eliminates the problem of tracing rays that do not contribute to the rendered image. Furthermore, the idea opens the door to numerous optimizations that speed up the process further.

This chapter discusses several ray tracing algorithms. First are the algorithms that form the basis of all ray tracing algorithms: ray-object intersection algorithms, light reflection and refraction formulae, the direct lighting algorithm, and the formulae for casting rays from a pinhole camera. The efficiency of the ray-object intersection algorithm is paramount due to the frequency with which it is used in all ray tracing algorithms. Thus, its efficiency is discussed later in the chapter. The classical ray tracing algorithm is discussed next. However, this algorithm has troubles accurately generating images for scenes that contain indirect lighting, and so several ray tracing algorithms that handle global illumination are also discussed.

Motion blurring is generally considered to be important when synthesizing high quality

images that contain moving objects. For instance, if an image were to contain two moving objects, one moving much faster than the other, but neither object were to be motion blurred, then there would be absolutely no way to determine which object was moving faster. Furthermore, there would be no way to determine that the objects are moving at all. In real life photography, motion blur arises when an object in the scene is moving at a high velocity with respect to the shutter speed of the camera. This motion while the film is exposed causes a blurring effect that image synthesis algorithms, striving for photorealism, must duplicate. Thus, how to add this ability to the ray tracing algorithms presented is also discussed in this chapter.

All of the algorithms used in ray tracing tend to be used tens, if not hundreds, of millions of times per image. Thus, the latter portion of this chapter is dedicated to efficiency considerations in ray tracing.

2.1 Ray-Object Intersections

One of the key elements of all ray tracing algorithms is the algorithm to determine at which point a given ray of light first contacts an object in the scene. This first ray-object intersection algorithm is employed many times for each ray that is cast from the camera into the scene being rendered. It is used every time the first intersection point of a ray of light is required, so that the the colour of light emitted in the opposite direction of the ray can be determined. It can also be utilized multiple times while calculating the colour of light emanating from a given surface point.

Fortunately, given algorithms to calculate the first intersection of a given ray with a given scene object, the basic first ray-object intersection algorithm is trivial. It simply calculates the distance, d, along the ray to the first ray-intersection with each object in the scene. Letting d_{min} be the smallest of these distances, the point d_{min} along the ray is the first ray-object intersection point of all objects in the scene.

2.2 Light Reflection and Refraction

While tracing a ray through a scene, if the first ray intersection is either a mirrored or transparent point on a surface, the ray needs to be reflected or refracted, as appropriate, to continue the trace until a non-mirrored and non-transparent object is intersected. Not doing this would result in mirrors that do not reflect an image, or glass windows that do not show the world beyond. Clearly, this is an unacceptable behaviour in an image synthesis algorithm that strives to create realistic images. Thus, this section reviews the models for reflecting and refracting light.

2.2.1 Mirror Reflecting Light

The law of reflection states that "the reflected ray lies in the plane of incidence and has an angle of reflection equal to the angle of incidence." [11] This law is illustrated in Figure 2.1, where $\boldsymbol{R}$ is the incident ray, $\boldsymbol{R}'$ the reflected ray, and $\boldsymbol{N}$ the normal to the surface.

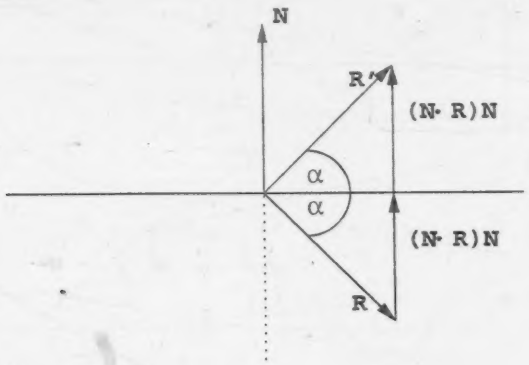

Figure 2.1: Reflection of a ray, $\boldsymbol{R}$.

As seen in Figure 2.1, the reflected ray, $\boldsymbol{R}'$, can be derived from the vector $\boldsymbol{R}$ by subtracting twice the projection of $\boldsymbol{R}$ onto the surface normal, $\boldsymbol{N}$. This yields the following formula for $\boldsymbol{R}'$:

$$\boldsymbol{R}' = \boldsymbol{R} - 2 \times (\boldsymbol{R} \cdot \boldsymbol{N})\boldsymbol{N}. \tag{2.1}$$

2.2.2 Refraction

The law of refraction states that "a refracted ray lies in the plane of incidence and has an angle of refraction that is related to the angle of incidence" by Equation 2.2. [11]

$$n' \sin\theta' = n \sin\theta. \tag{2.2}$$

Here each of the symbols n and n' is a dimensionless constant called the *index of refraction* and is associated with the mediums on either side of the interface.

By consulting Figure 2.2, a formula for expressing the refracted ray, $\boldsymbol{R}'$, in terms of the surface normal, $\boldsymbol{N}$, and the incident ray, $\boldsymbol{R}$, can be derived. Since the vectors $\boldsymbol{N}$ and $\boldsymbol{B}$ are orthogonal unit vectors, they form a basis of the plane of refraction. Thus, the vectors $\boldsymbol{R}$ and $\boldsymbol{R}'$ can be expressed in terms of this basis as:

$$\boldsymbol{R} = \boldsymbol{B} \sin\theta - \boldsymbol{N} \cos\theta, \tag{2.3}$$

$$\boldsymbol{R}' = \boldsymbol{B} \sin\theta' - \boldsymbol{N} \cos\theta'. \tag{2.4}$$

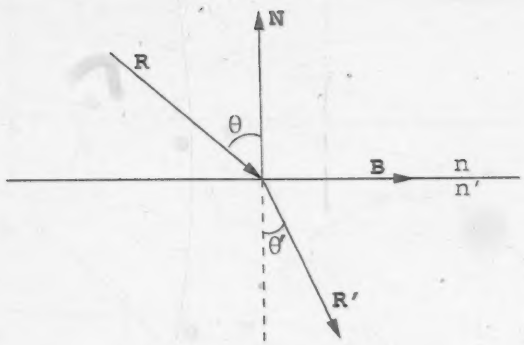

Figure 2.2: Refraction of a ray, $\boldsymbol{R}$.

Since $\sin\theta$, $\cos\theta$, $\boldsymbol{R}$, and $\boldsymbol{N}$ are known quantities, a formula for $\boldsymbol{B}$ can be derived from Equation 2.3.

$$\begin{aligned} B &= \frac{\boldsymbol{R} + \boldsymbol{N}\cos\theta}{\sin\theta}, \\ &= \frac{\boldsymbol{R} - (\boldsymbol{R}\cdot\boldsymbol{N})\boldsymbol{N}}{\sin\theta}. \end{aligned} \tag{2.5}$$

Equation 2.6 is derived from Equation 2.2, and the formula $\sin^2\theta' + \cos^2\theta' = 1$.

$$\cos^2\theta' = 1 - \left(\frac{n'}{n}\right)^2 (1 - \cos^2\theta). \tag{2.6}$$

Finally, combining Equations 2.2, 2.4, 2.5, and 2.6 arrives at the equation for the refracted ray:

$$\boldsymbol{R'} = \frac{n}{n'}(\boldsymbol{R} - (\boldsymbol{R}\cdot\boldsymbol{N})\boldsymbol{N}) - \boldsymbol{N}\sqrt{1 - \left(\frac{n}{n'}\right)^2 (1 - (\boldsymbol{R}\cdot\boldsymbol{N}))^2}. \tag{2.7}$$

2.3 Direct Lighting

Once the first intersection point of a ray with the objects in the scene has been calculated, the shade of light emanating from the point in the opposite direction to the ray's direction has to be calculated. This will give the colour of the light ray that is presently being traced through the scene, which helps determine the colour of the pixel the ray originated from. Direct lighting is due to lighting directly from light sources without interactions with the environment. Simply speaking, if, while standing on the point being shaded, there is a direct line of sight to a light source, then that light source contributes to the direct lighting of the point.

In the fairly common case that there is more than one light source in the scene, the direct lighting of a point is the sum of the direct lighting from every light in the scene. Thus, once

able to calculate the lighting from a single point light the direct lighting from all lights in the scene can be calculated. Given a point light at the point $\boldsymbol{L}$, the remainder of this section concerns itself with calculating the shading of the point $\boldsymbol{x}$ with outward facing normal $\boldsymbol{N}$. The lighting models presented in this section are a simplification of true, physical, lighting models.

2.3.1 Visibility of a Light Source

Before discussing how to use visible lights when calculating the illuminance of a point, $\boldsymbol{x}$, it is important to mention how to determine if a given point light at the point $\boldsymbol{L}$ is visible from $\boldsymbol{x}$. Let $\boldsymbol{N}$ be the surface normal at $\boldsymbol{x}$.

Let $\boldsymbol{l} = \boldsymbol{L} - \boldsymbol{x}$. Since $\boldsymbol{x}$ lies on a surface, if $\boldsymbol{l} \cdot \boldsymbol{N} > 0$ then the angle between $\boldsymbol{N}$ and $\boldsymbol{l}$ is less than 90 degrees. That is, $\boldsymbol{L}$ lies above the tangent plane through $\boldsymbol{x}$. If this angle is greater than 90 degrees, then $\boldsymbol{L}$ lies below the tangent plane through $\boldsymbol{x}$, and so cannot possibly be visible from $\boldsymbol{x}$. This observation provides a very efficient visibility test.

Once this visibility test has been performed, what remains is to determine if any objects intersect the line segment $\boldsymbol{x}$-$\boldsymbol{L}$. If any do, then $\boldsymbol{x}$ is in a region of shadow relative to the light source. To perform this portion of the visibility test the first ray-intersection algorithm presented above can be used.

2.3.2 Diffuse Surfaces

To calculate the diffuse, or matte, contribution to the light emanating from a surface point *Lambert's Law* (Equation 2.8) can be used. This law serves as a good approximation to how matte surfaces react to lighting:

$$I_d = I_{\mathcal{L}} k_d \cos\theta, \tag{2.8}$$

where $I_{\mathcal{L}}$ is the colour of the light source, k_d the diffuse reflectance of the surface, and θ the angle between the surface normal at the illuminated point and the direction from the illuminated point to the light source.

An equation to calculate I can be derived using Lambert's Law. Consider Figure 2.3. Let $\boldsymbol{R} = \boldsymbol{o} + k \times \boldsymbol{d}$ be the ray being traced that intersects the surface at the point x. Furthermore, let $\boldsymbol{N}$ be the surface normal at $\boldsymbol{x}$, and $\boldsymbol{l}$ be the unit length direction vector from $\boldsymbol{x}$ to the light source. Since $\boldsymbol{N}$ and $\boldsymbol{l}$ are unit vectors, the cosine of the angle between $\boldsymbol{N}$ and $\boldsymbol{l}$ is:

$$\cos\theta = \boldsymbol{N} \cdot \boldsymbol{l}.$$

Plugging this into Lambert's Law arrives at:

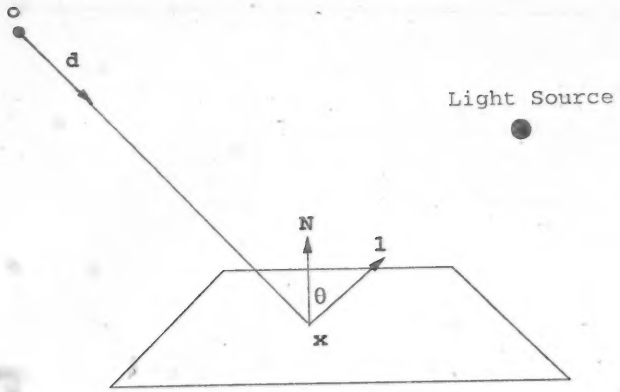

Figure 2.3: Geometry for diffuse lighting calculation.

$$I_d = I_{\mathcal{L}} k_d (\boldsymbol{N} \cdot \boldsymbol{l}) . \tag{2.9}$$

Notice that Equation 2.9 does not actually use the components of the ray, $\boldsymbol{R}$, anywhere. This means that the diffuse contribution to the emitted light at $\boldsymbol{x}$ is independent of the angle at which the surface is viewed. Considering a matte surface in real life, this property actually makes sense. The shade of a real life matte surface does not vary with respect to the angle at which it is viewed.

2.3.3 Specular Surfaces

Specular surfaces that are not perfect mirrors still reflect light, but the reflected light is not as focused as it would be when reflected from a perfect mirror. To account for this, a specular reflection model is added to the direct lighting calculations that are used on specular surfaces that are not perfect mirrors. The Phong reflection model [8, 16] relates the intensity of the reflected light to the angle between the mirror reflected viewing direction and the direction to the light source.

Consider Figure 2.4, which illustrates the vectors and angles involved in the Phong reflection model. In this figure, the ray being traced is represented as $\boldsymbol{o} + k \times \boldsymbol{d}$ (where $\boldsymbol{d}$ is a unit length vector), $\boldsymbol{x}$ is the intersection point of the ray with the surface, $\boldsymbol{N}$ is the surface normal at $\boldsymbol{x}$, $\boldsymbol{l}$ is the unit direction vector from $\boldsymbol{x}$ to the light source, $\boldsymbol{H}$ is the mirror reflection direction of the ray, and α is the angle between $\boldsymbol{H}$ and $\boldsymbol{l}$. The Phong reflection model also uses a surface dependent constant, n_s, called the specular exponent. This constant typically has a value in the range $[1, 200]$[16]. The Phong reflection model states that the amount of incident light specularly reflected, in the direction opposite to the incoming ray, is:

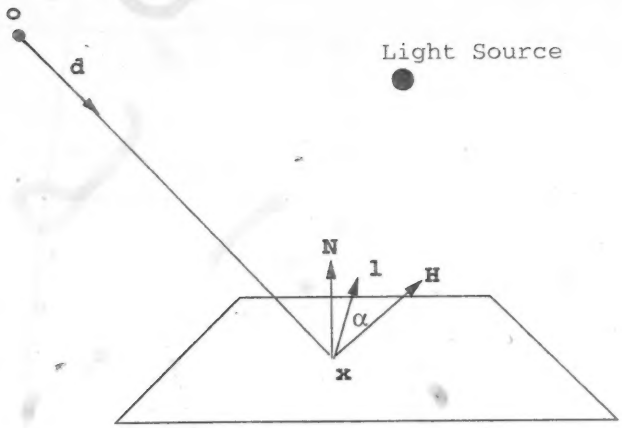

Figure 2.4: Geometry for specular lighting calculation.

$$I_s = I_{\mathcal{L}} k_s \cos^{n_s} \alpha, \tag{2.10}$$

where $I_{\mathcal{L}}$ is the intensity of the light source, and k_s is the specular reflection colour of the surface at the point of reflection (the point $\boldsymbol{x}$). Since both $\boldsymbol{H}$ and $\boldsymbol{l}$ are unit length vectors, the equation can be rewritten as:

$$I_s = I_{\mathcal{L}} k_s \left(\boldsymbol{H} \cdot \boldsymbol{l}\right)^{n_s}. \tag{2.11}$$

The effects of this equation, on a rendered object, are to add highlights to the object if it has specular properties. As the value of n_s is increased, the highlight becomes smaller and more focused. In fact, if n_s approaches ∞, the incident light is reflected as if by a perfect mirror.

2.3.4 Shading

The elements of the diffuse shading model, the Phong reflection model, the mirror reflection model, and the refraction model can now be combined into a single equation. Given the point $\boldsymbol{x}$ with surface normal $\boldsymbol{N}$, the colour of the light, $I(\boldsymbol{x}, \boldsymbol{\omega})$, emanating from $\boldsymbol{x}$ in the direction $\boldsymbol{\omega}$ is:

$$I(\boldsymbol{x}, \omega) = k_d(\boldsymbol{x}) \left[\sum_{\mathcal{L} \in \mathcal{V}(\boldsymbol{x})} I_{\mathcal{L}} \left(\boldsymbol{N} \cdot \boldsymbol{L}_{\mathcal{L}}\right) \right] + k_s(\boldsymbol{x}) \left[\sum_{\mathcal{L} \in \mathcal{V}(\boldsymbol{x})} I_{\mathcal{L}} \left(\boldsymbol{L}_{\mathcal{L}} \cdot \boldsymbol{H}\right)^{n_s(\boldsymbol{x})} \right] + k_r(\boldsymbol{x}) I_r + k_t(\boldsymbol{x}) I_t \tag{2.12}$$

where the variables are defined as in Table 2.1.

$\mathcal{V}(\boldsymbol{x})$	The set of all lights visible from $\boldsymbol{x}$.
$k_d(\boldsymbol{x})$	The diffuse colour of the material at point $\boldsymbol{x}$. This value is also referred to as the pigment, Lambertian, or matte colour of the material at $\boldsymbol{x}$.
$k_s(\boldsymbol{x})$	The specular reflection colour of the material at point $\boldsymbol{x}$.
$n_s(\boldsymbol{x})$	The specular reflection exponent of the material at the point $\boldsymbol{x}$. $n_s(\boldsymbol{x}) > 0$, and controls how focused the specular highlight is. Typically, n_s is in the range $[1, 200]$.
$k_r(\boldsymbol{x})$	The mirror reflection colour of the material at point $\boldsymbol{x}$.
$k_t(\boldsymbol{x})$	The transmission, or transparency, colour of the material at point $\boldsymbol{x}$.
$\boldsymbol{L}_{\mathcal{L}}$	Let $\boldsymbol{P}$ be the position of light $\mathcal{L}$, then $\boldsymbol{L}_{\mathcal{L}} = \frac{\boldsymbol{P}-\boldsymbol{x}}{\|\boldsymbol{P}-\boldsymbol{x}\|}$.
$\boldsymbol{H}$	The mirror reflection of $\boldsymbol{\omega}$ about $\boldsymbol{N}$.
$I_{\mathcal{L}}$	The colour of light source $\mathcal{L}$.
I_r	The colour of the light from the mirror reflection direction (non-zero only if the material at point $\boldsymbol{x}$ is a mirror).
I_t	The colour of the light from the transmission/refraction direction (non-zero only if the material at point $\boldsymbol{x}$ is transparent).

Table 2.1: Variables and constants of Equation 2.12.

2.3.5 Direct Lighting from Area Light Sources

Another type of light source that can be included in a ray tracer is the area light. Unlike a point light, which emanates light in all directions from a single point, an area light emanates light over an extended area. For example, a fluorescent rod light is a cylindrical area light. In fact, all lights in the real world are area lights. The point light was introduced to computer graphics as an approximation to a light bulb that is much easier to work with.

An area light can be thought of as a densely packed formation of an infinite number of point lights. Unfortunately, thinking of an area light in this manner does not work well in ray tracing algorithms because it would be impossible to base a single lighting calculation on an infinite number of point lights. Thus, in the interest of efficiency, an area light is instead treated as a uniformly random distribution of a finite number of point lights over the surface of the area light. The sum of the luminance of these m random point lights should equal the luminance of the area light. Thus, if the area light has luminance L, each of the m point lights is assigned a luminance of $\frac{L}{m}$. These m point lights can now be included when calculating the colour of light emanating from a point using Equation 2.12.

2.4 The Camera

As with all image synthesis techniques, ray tracing requires the definition of a camera. Shirley [23] describes a simple pinhole camera for this purpose. Shirley's camera uses an orthonormal basis, which specifies a local coordinate system for the camera, an image plane, and a viewing rectangle within the image plane, which is the region of the image plane that

makes up the final image. To use his camera model the variables seen in Figure 2.5 are defined as follows:

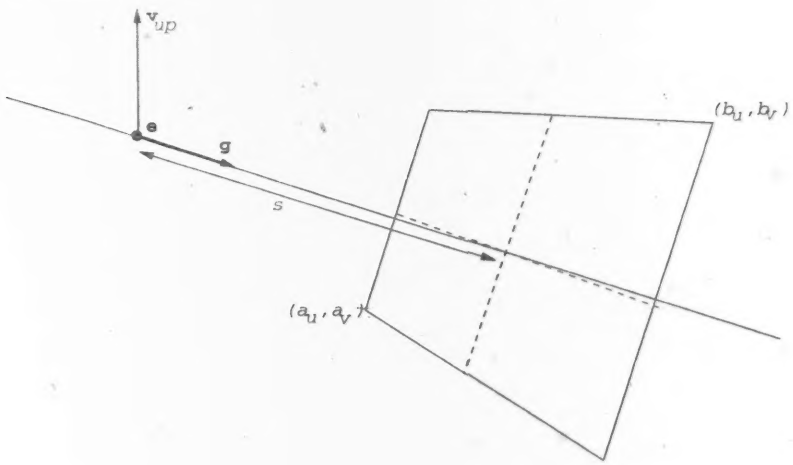

Figure 2.5: The pinhole camera.

- $\boldsymbol{e}$, the 3D position of the pinhole, or eyepoint;
- $\boldsymbol{g}$, the 3D gaze, or view direction, vector;
- $\boldsymbol{v_{up}}$, the 3D view-up vector. This vector points in the general direction from the eye to the 'top' of the camera;
- s, the distance from the eye to the viewplane;
- a_u, a_v, b_u, b_v, the viewplane-local (uv) coordinates of the lower-left and upper-right corners of viewing rectangle;
- n_x, n_y, the image width and height, respectively, in pixels.

The vectors $\boldsymbol{g}$, and $\boldsymbol{v_{up}}$, are then used to create the orthonormal basis $\boldsymbol{w}$, $\boldsymbol{u}$, $\boldsymbol{v}$ as follows:

$$\begin{aligned} \boldsymbol{w} &= -\frac{\boldsymbol{g}}{||\boldsymbol{g}||}, \\ \boldsymbol{u} &= \frac{\boldsymbol{v_{up}} \times \boldsymbol{w}}{||\boldsymbol{v_{up}} \times \boldsymbol{w}||}, \\ \boldsymbol{v} &= \boldsymbol{w} \times \boldsymbol{u}. \end{aligned}$$

This basis can be used to generate a ray, $\boldsymbol{P(k)}$, through the center of pixel (i, j) in the image:

$$\boldsymbol{P(k)} = e + k((a_u + (b_u - a_u)\frac{i + 0.5}{n_x - 1})\boldsymbol{u} + (a_v + (b_v - a_v)\frac{j + 0.5}{n_y - 1})\boldsymbol{v} - s\boldsymbol{w}). \qquad (2.13)$$

This equation calculates the ray, $\boldsymbol{P(k)}$, using a standard change of basis equation. In the coordinate system $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}\}$, the image plane passes through the point $< 0, 0, -s >$ and has surface normal $\boldsymbol{w}$. Furthermore, the viewing rectangle has lower-left and upper-right coordinates $< a_u, a_v, -s >$ and $< b_u, b_v, -s >$, respectively. Each of the $n_x \times n_y$ pixels in the image being generated is a sub-rectangle of this larger rectangle. The rectangle for pixel (i, j) has lower-left and upper-right corners $< a_u + (b_u - a_u)\frac{i}{n_x - 1}, a_v + (b_v - a_v)\frac{j}{n_y - 1}, -s >$ and $< a_u + (b_u - a_u)\frac{i+1}{n_x - 1}, a_v + (b_v - a_v)\frac{j+1}{n_y - 1}, -s >$, respectively. Thus, the center of pixel (i, j) is at the point $< a_u + (b_u - a_u)\frac{i+0.5}{n_x - 1}, a_v + (b_v - a_v)\frac{i+0.5}{n_y - 1}, -s >$ in the $\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}\}$ coordinate system.

However, if only a single ray is cast through each pixel of the image then the image would suffer from aliasing artifacts. That is, object edges that do not run either horizontally or vertically in the image will appear jagged and pixelated. To compensate for this, all ray tracing algorithms cast more than one ray through each pixel. Doing so anti-aliases the jagged edges, making the image more pleasing to the eye. In order to cast more than one ray through each pixel the ability to cast a ray through a location other than the center of the pixel is required. This is done with the following formula:

$$\boldsymbol{P(k)} = \boldsymbol{e} + k((a_u + (b_u - a_u)\frac{i + \epsilon_0}{n_x - 1})\boldsymbol{u} + (a_v + (b_v - a_v)\frac{j + \epsilon_1}{n_y - 1})\boldsymbol{v} - s\boldsymbol{w}) \tag{2.14}$$

where ϵ_0, and ϵ_1 are uniformly distributed random numbers in the range $[0, 1]$. This equation is derived in exactly the same way as Equation 2.13, except instead of casting the ray through the center of the pixel, it casts the ray through a uniformly random point within the pixel.

2.5 The Classical Ray Tracing Algorithm

As with all image synthesis techniques, the classical ray tracing algorithm begins with the geometric objects, lights, and camera properties defined by the user of the system. It generates an image by casting N_s randomly generated rays from the camera through each pixel in the final image into the scene. The goal is to generate an n_x by n_y pixel RGB image of the scene that has been described. For each ray, it then determines the first object it intersects and calculates the colour of light emanating from the point of intersection along the ray using Equation 2.12. Only if the intersection point, $\boldsymbol{x}$, is a perfect mirror, is the part of the equation that adds light from the mirror reflection direction $(k_r(\boldsymbol{x})I_r)$ utilized. In this case, the mirror reflected ray is recursively traced into the scene and the colour of that ray is used as the I_r term in the equation. Similarly, only if the intersection point, $\boldsymbol{x}$, is transparent, is the part of Equation 2.12 that adds light from through the surface $(k_t(\boldsymbol{x})I_t)$ utilized. In this case, the refracted ray is recursively traced and the colour of that ray is used as the I_t term in the equation.

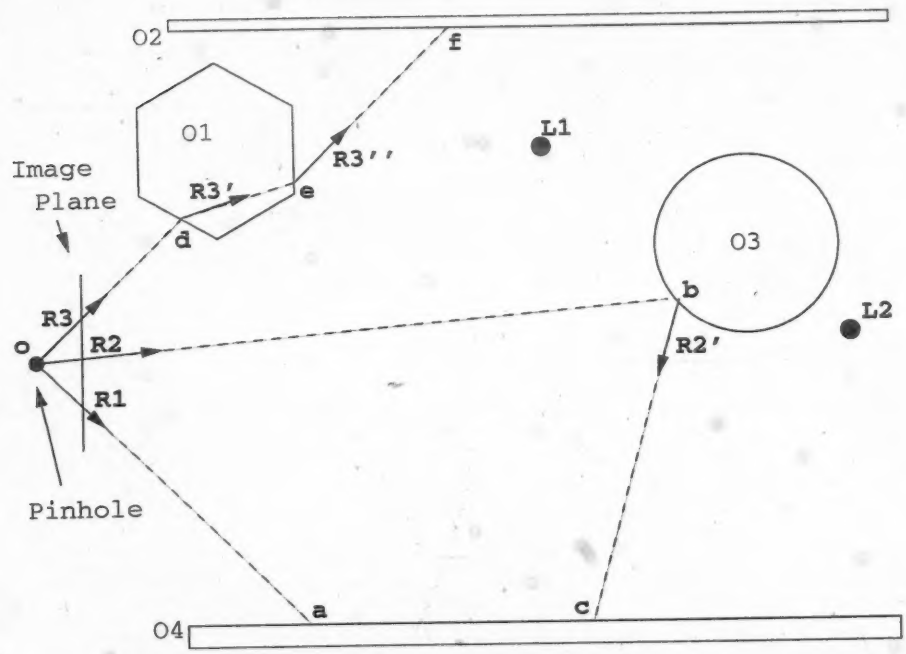

Figure 2.6: A scene to demonstrate the classical ray tracing algorithm.

As an example, consider the two dimensional scene depicted in Figure 2.6. In this scene, object O1 is a transparent hexagon, object O3 is a circular perfect mirror, and objects O2 and O4 are matte rectangles. The point $\boldsymbol{o}$ denotes the eyepoint (the pinhole of the pinhole camera), and the points $\boldsymbol{L1}$ and $\boldsymbol{L2}$ denote light sources. The points $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$, $\boldsymbol{e}$, and $\boldsymbol{f}$ are the intersection points of the light rays with the objects in the scene.

The ray $\boldsymbol{R1}$ is traced into the scene, and intersects rectangle O4 at $\boldsymbol{a}$. Since object O4 is neither a perfect mirror nor transparent, the classical ray tracing algorithm does not recursively trace any rays while calculating the colour of ray $\boldsymbol{R1}$. When calculating the direct lighting at $\boldsymbol{a}$, both lights $\boldsymbol{L1}$ and $\boldsymbol{L2}$ are used because they are both visible from $\boldsymbol{a}$.

Next, the ray $\boldsymbol{R2}$ is traced into the scene, and intersects circle O3 at $\boldsymbol{b}$. Since object O3 is a perfect mirror, but not transparent, the classical ray tracing algorithm will recursively trace the mirror reflected ray $\boldsymbol{R2'}$ but will not recursively trace a refracted ray. The recursively traced ray $\boldsymbol{R2'}$ intersects rectangle O4 at $\boldsymbol{c}$ and is the direct lighting at $\boldsymbol{c}$ is calculated using both lights $\boldsymbol{L1}$ and $\boldsymbol{L2}$. The colour of ray $\boldsymbol{R2'}$ is used in the direct lighting calculation for $\boldsymbol{R2}$, but neither of the light sources are used because they are not visible from $\boldsymbol{b}$.

Lastly, the ray $\boldsymbol{R3}$ is traced into the scene and intersects hexagon O1 at $\boldsymbol{d}$. The hexagon is transparent, but not a perfect mirror, so the refracted ray $\boldsymbol{R3'}$ is recursively traced to get the refracted light colour but no mirror reflected rays are traced. While recursively tracing the ray $\boldsymbol{R3'}$, it refracts through the interface of the hexagon once more before striking

rectangle O2 at $\boldsymbol{f}$. The direct lighting at $\boldsymbol{f}$ is calculated using light $\boldsymbol{L1}$, but not $\boldsymbol{L2}$ ($\boldsymbol{L2}$ is not visible), and the value is used when calculating the direct lighting at $\boldsymbol{e}$, which is in turn used to calculate the direct lighting at $\boldsymbol{d}$. Neither of the light sources is visible from $\boldsymbol{d}$, so the direct lighting of the point $\boldsymbol{d}$ is calculated using only the colour of the refracted light.

In pseudocode, the classical ray tracing algorithm can be expressed as:

```
function ClassicalRayTrace
    for each pixel, (i,j), in the output image
        pixelColour ← black
        for k = 1...N_s
            ray ← random ray, from the camera, through pixel (i,j).
            pixelColour += trace(ray)
        endfor
        image pixel (i,j) ← pixelColour / N_s
    endfor


function trace(ray)
    object ← firstIntersection(ray)
    x ← intersection point of ray with object
    colour ← shade(ray, object, x)
    if object is a mirror
        colour += k_r(x) trace(reflected ray)
    endif
    if object is transparent
        colour += k_t(x) trace(refracted ray)
    endif
    return colour
```

where $k_r(\boldsymbol{x})$ and $k_t(\boldsymbol{x})$ are the mirror and transmission colours at $\boldsymbol{x}$, respectively.

The classical ray tracing algorithm can be used to generate images for simple scenes in which all of the items of interest are directly illuminated by the lights in the scene; such as a scene of the mountains during a bright sunny day, with no clouds in the sky. However, the algorithm falls short when generating images of a scene illuminated by indirect lighting; such as a room that is illuminated by a flashlight directed at a mirror. For instance, in the example illustrated above, which uses Figure 2.6, the point $\boldsymbol{f}$ was directly illuminated by light source $\boldsymbol{L1}$ but not by light source $\boldsymbol{L2}$. In the real world, some amount of light from $\boldsymbol{L2}$ would reflect off of object O4 to illuminate $\boldsymbol{f}$. This indirect illumination would not be as intense as $\boldsymbol{f}$ being directly illuminated by $\boldsymbol{L2}$, but would still be visibly noticeable if it were added.

2.6 Global Illumination

When light is shone on a surface, it reflects and scatters off of the surface. The effect is that light reaches regions where there is no direct line of sight to any light source. Thus, if an image synthesis algorithm, such as ray tracing, is to produce realistic images of the real world, it must be able to simulate this indirect illumination as well as direct, line of sight, illumination. The combination of these two types of illumination, direct and indirect, is typically referred to as global illumination.

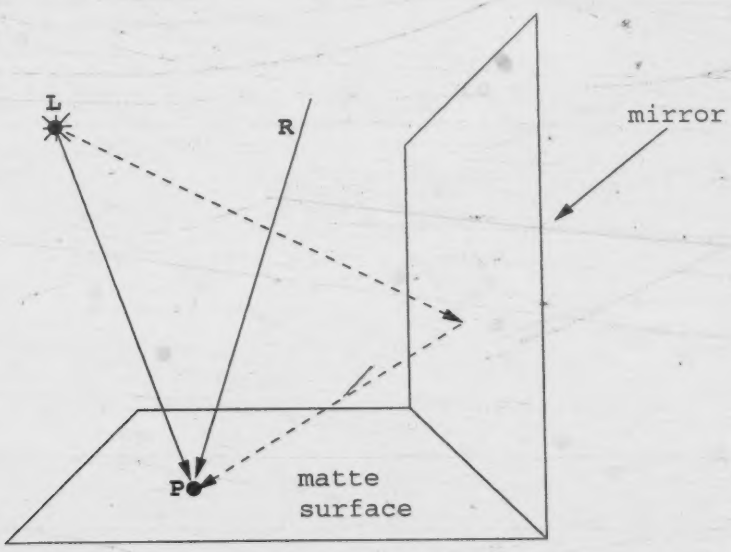

Figure 2.7: Direct and indirect illumination of a point P.

The classical ray tracing algorithm only utilizes direct illumination when generating an image for a given scene. This limits the realism that can be achieved by the algorithm.

Consider the scene in Figure 2.7 that shows the illumination of a point, P, being calculated for the ray R, by a point light at L. The solid line indicates the direct lighting contribution of the point light L to the lighting of the point P. However, the dashed line indicates a path of light that originates at L reflects off of the mirror, and arrives at the point P. This path of light should contribute to the illumination of the point P, but the direct lighting model used in the classical ray tracing algorithm is incapable of finding this light path.

2.6.1 Distributed Ray Tracing

As a solution to this problem, Cook et al. [7] present the distributed ray tracing algorithm as an extension to the classical ray tracing algorithm to support global illumination. When light strikes a diffuse surface, it is scattered by the surface. Because of this scattering

the distributed ray tracing algorithm adds diffuse reflections to the classical ray tracing algorithm.

Diffuse light scattering is added to the classical ray tracing algorithm by emitting a number of rays at random orientation from the surface at $\boldsymbol{x}$ every time that the direct lighting of $\boldsymbol{x}$ is calculated. Each of these rays are traced recursively to determine its colour, which is then added to the illumination of $\boldsymbol{x}$. To add the light from each of these rays to the illumination of point $\boldsymbol{x}$ the shading Equation 2.12 is utilized and each ray is treated as a visible light source.

For example, consider the two dimensional scene used in demonstrating the classical ray tracing algorithm (Figure 2.6). Specifically, consider the calculations involved in calculating the colour of light emitted from the point $\boldsymbol{a}$. In the classical ray tracing algorithm, the light colour is determined solely from the direct, line of sight, contributions of light sources $\boldsymbol{L1}$ and $\boldsymbol{L2}$. In the distributed ray tracing algorithm of Cook et al. this same direct lighting is calculated, but N_f rays, $\boldsymbol{R1}'_1, \boldsymbol{R1}'_2, \ldots, \boldsymbol{R1}'_{N_f}$, are also generated originating at $\boldsymbol{a}$ each in a random direction in the half-circle around the surface tangent at $\boldsymbol{a}$. Some of these N_f rays will intersect nothing, but some of them will eventually intersect the scene at diffuse points; each ray may be traced through one or more refractions or mirror reflections, in the same way as in the classical ray tracing algorithm, before shooting outside of the scene or reaching its diffuse intersection point. The light colour of each of these rays will then be calculated using direct lighting and by casting another N_f random rays in the same way. For the rays that shoot outside of the scene, the light colour of the ray is set to a user-defined background colour. Only when a user-defined maximum recursive depth is reached does the algorithm stop generating rays to be traced recursively. Let the calculated light colour of the ray $\boldsymbol{R1}'_i$ be $I_{\boldsymbol{R1}'_i}$, then the following is added to the illumination of $\boldsymbol{a}$:

$$\frac{1}{N_f}\left[k_d(\boldsymbol{x})I_{\boldsymbol{R1}'_i}\left(\boldsymbol{N}\cdot\boldsymbol{R1}'_i.d\right) + k_s(\boldsymbol{x})I_{\boldsymbol{R1}'_i}\left(\boldsymbol{H}\cdot\boldsymbol{R1}'_i.d\right)^{n_s(\boldsymbol{x})}\right],$$

where $\boldsymbol{H}$ is the mirror reflection of $\boldsymbol{R1}$, and the other variables are defined as in Table 2.1.

In pseudocode, the distributed ray tracing algorithm is:

```
function DistributedRayTrace
    for each pixel, (i, j), in the output image
        pixelColour ← black
        for k = 1 ... N_s
            ray ← random ray, from the camera, through pixel (i, j).
            pixelColour += trace(ray, 0)
        endfor
        image pixel (i, j) ← pixelColour / N_s
    endfor
```

```
function trace(ray, depth)
    object ← firstIntersection(ray)
    x ← intersection point of ray with object
    colour ← shade(ray, object, x)
    if object is a mirror
        colour += k_r(x) trace(reflected ray, depth)
    endif
    if object is transparent
        colour += k_t(x) trace(refracted ray, depth)
    endif
    ---------------------------------------
    if (depth < MAX_DEPTH)
        indirect_colour ← black
        for i = 1...N_f do
            R ← random ray from x
            R_colour ← trace(R, depth+1)
            indirect_colour + = addLight(R_colour)
        endfor
        colour + = indirect_colour
    endif
    ---------------------------------------
    return colour
```

Cook et al. add the region of code outlined above to the classical ray tracing algorithm to form the distributed ray tracing algorithm. This code ensures that the recursive depth does not exceed the maximum recursive depth, MAX_DEPTH, before recursively tracing N_f random rays from x. The indirect illumination calculated by these rays is then added to the illumination at x.

Although the distributed ray tracing algorithm is able to simulate the effects of global illumination in a scene, it does so at a very high price. The algorithm suffers from an exponential growth in running time as the maximum recursive depth, MAX_DEPTH, is increased to improve accuracy of the algorithm. Furthermore, increasing N_f by as little as one causes the number of rays traced by the ray tracer to increase by $O(N_f^{MAX_DEPTH-1})$. Additionally, due to the $\frac{1}{N_f}$ scaling factor used when adding the lighting contribution of one of the randomly generated diffusely reflected rays, rays traced at each depth only add $\left(\frac{1}{N_f}\right)^{\text{depth}}$ of their illuminance to the top-level ray colour. Thus, the larger the MAX_DEPTH value used, the less the rays at the added depth will add to the illumination of the top-level ray.

Clearly, if the user is not concerned with efficiency this algorithm would be sufficient for ray tracing nearly any scene imaginable. Unfortunately, it is only viable for small images that are traced to a fairly small depth.

2.6.2 Irradiance Caching

Ward [26] introduces an algorithm to improve performance when using the distributed ray tracing algorithm to render a scene. This algorithm, the irradiance cache, caches the results of diffuse indirect lighting calculations, as they are calculated, to be used when calculating the indirect illumination of other points in the scene.

The distributed ray tracing algorithm is modified as follows. Prior to calculating the indirect lighting of a given surface point, $\boldsymbol{x}$, by recursively casting N_f rays, Ward's algorithm searches the irradiance cache for a cached irradiance value suitable to be used at $\boldsymbol{x}$. If a suitable value can be found, then the cached irradiance value is used as the irradiance at $\boldsymbol{x}$ and no recursive rays will be cast to calculate the indirect illumination. Otherwise, if a suitable value cannot be found, the indirect illumination of $\boldsymbol{x}$ is calculated by recursively casting N_f rays as in distributed ray tracing. Once the indirect illumination for $\boldsymbol{x}$ has been calculated, the the value is stored in the irradiance cache for later use.

This algorithm speeds up the distributed ray tracing algorithm by removing many of the recursive indirect lighting calculations performed in the distributed ray tracing algorithm. Some of these computationally intense recursive calculations are replaced by a faster cache lookup, resulting in substantial speedup to the algorithm.

2.6.3 The Rendering Equation

Kajiya [17] presents an equation that now forms the basis for all global illumination algorithms. His equation states that the illumination of a point $\boldsymbol{x}$ by a point $\boldsymbol{x}'$ is the sum of the emitted light at $\boldsymbol{x}'$ and the light scattered from $\boldsymbol{x}'$ toward $\boldsymbol{x}$:

$$I(\boldsymbol{x},\boldsymbol{x}') = g(\boldsymbol{x},\boldsymbol{x}')\left[e(\boldsymbol{x},\boldsymbol{x}') + \int_S \rho(\boldsymbol{x},\boldsymbol{x}',\boldsymbol{x}'')I(\boldsymbol{x}',\boldsymbol{x}'')d\boldsymbol{x}''\right], \tag{2.15}$$

where:

$I(\boldsymbol{x},\boldsymbol{x}')$	is the intensity of light passing from $\boldsymbol{x}'$ to $\boldsymbol{x}$;
$g(\boldsymbol{x},\boldsymbol{x}')$	$= \begin{cases} 1 \text{ if } \boldsymbol{x} \text{ is visible from } \boldsymbol{x}' \\ 0 \text{ otherwise} \end{cases}$;
$e(\boldsymbol{x},\boldsymbol{x}')$	the intensity of the emitted light from $\boldsymbol{x}'$ to $\boldsymbol{x}$;
$\rho(\boldsymbol{x},\boldsymbol{x}',\boldsymbol{x}'')$	the intensity of light scattered from $\boldsymbol{x}''$, by a surface at $\boldsymbol{x}'$, to $\boldsymbol{x}$;

S is the union of all surfaces in the scene.

This equation was developed as an approximation to the Maxwell equations for electromagnetics to provide a unified context that can be used to study a wide variety of rendering algorithms [17]. For example, the distributed ray tracing algorithm approximates the rendering equation by discreetly sampling the integral as:

$$\int_S \rho(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}'') I(\boldsymbol{x}', \boldsymbol{x}'') d\boldsymbol{x}'' \approx \frac{1}{N_f} \sum_{i=1}^{N_f} [\rho(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}_i) I(\boldsymbol{x}', \boldsymbol{x}_i)] \,. \tag{2.16}$$

where $\boldsymbol{x}_i$ is the intersection of the i^{th} randomly generated ray with the scene.

2.6.4 Path Tracing

As part of Kajiya's work on the rendering equation [17], he also proposes a new ray tracing algorithm as a solution to the equation. This new algorithm, called path tracing, is actually a simplification of the distributed ray tracing algorithm that removes both the exponential growth and the diminishing returns problems. In exchange, however, a substantially larger value of N_s is required in the path tracing algorithm than is required in the distributed ray tracing algorithm to render images with similar quality. Fortunately, since the path tracing algorithm does not have an exponential growth problem, it will generally outperform the distributed ray tracing algorithm.

The difference between the path tracing algorithm and the distributed ray tracing algorithm is that in the path tracing algorithm $N_f = 1$, and in the distributed ray tracing algorithm $N_f \geqslant 1$. Beyond this difference, the two algorithms are identical. Since $N_f = 1$ in path tracing, there is no exponential growth, and no diminishing returns problem as in the distributed ray tracing algorithm. In exchange, the number of rays cast per pixel in the image needs to be increased. In images with slowly varying lighting, a few hundred rays per pixel is sufficient. However, if the lighting in the scene varies drastically, and rapidly, several thousand rays per pixel may be required.

2.6.5 Backwards Ray Tracing

One of the potential problems with the distributed ray tracing and the path tracing algorithm is that some lighting conditions, such as caustics, are quite difficult to generate with the two algorithms. Caustics are generated when light is redistributed. For example, the bright spots of light in the bottom of a swimming pool are caustics. Figure 2.8 is an example of a caustic formed by a metallic ring. For the distributed ray tracing and path tracing algorithms to duplicate these effects would require either very careful generation of the recursive rays such that the rays will reflect, or refract, into the light source causing

the caustics, or they would have to use a large enough number of rays to grant a very high probability of reflecting, or refracting, into the light source.

Figure 2.8: Image of a caustic formed by light shinning on a metallic ring.

As a simplification of this process, to allow ray tracing algorithms to generate caustics, Arvo [2] proposes a two-pass ray tracing algorithm called backwards ray tracing. This algorithm constructs an illumination map in the first pass, and renders the image, using the constructed illumination map for lighting calculations, in the second pass. Arvo's first implementation of the backwards ray tracing algorithm utilizes a texture on each surface, that captures the global illumination incident on the surface, for an illumination map [28]. However, this implementation accounts for only diffuse reflection and so is not a complete implementation of the backwards ray tracing algorithm. The first full implementation of the backwards ray tracing algorithm is attributed [27] to Chattopadhyay and Fujimoto [5].

To construct the illumination map, a number of rays are cast from the light sources into the scene. The rays are traced through refractions, and mirror reflections, until they intersect a diffuse surface. At that point, the intersection and light intensity is recorded in the illumination map. Rays of light that hit a diffuse surface directly are ignored.

During the second pass of the algorithm, either the distributed ray tracing or path tracing algorithm is used to render the scene. However, whenever a lighting calculation is performed, the illumination record located in the illumination map is also used to calculate lighting.

2.6.6 Photon Mapping

Photon mapping is an implementation of the backwards ray tracing method that has recently been popularized by Henrik Wann Jensen [14, 15]. The photon mapping algorithm actually utilizes two illumination maps. One illumination map, the caustics photon map, only stores photons that directly contribute to caustics, and the other, the global photon map, contains the remainder of the photons. A photon directly contributes to caustics only if it is cast from a light source, refracts, or mirror reflects, off one or more surfaces and then is deposited on a diffuse surface.

In the first pass of the photon mapping algorithm, the global photon map is created by casting a user-defined number, N_p, of light particles (photons) into the scene. When a photon intersects a perfect mirror or a refractive surface, the photon is reflected or refracted, respectively. But, when the photon intersects a diffuse surface, a probability that the photon will be diffusely reflected is calculated. If it is decided to diffusely reflect the photon, then the photon is diffusely reflected and tracing continues. On the other hand, if the photon is not diffusely reflected, the photon is stored into the photon map at the intersection point.

Creating the caustics photon map is done in much the same manner as the global photon map, with a few exceptions. When a photon is cast from a light source, it is cast only toward an object that may produce a caustic. For instance, objects such as a concave mirror, a glass lens, or a surface of water will produce caustics. Secondly, when the photon intersects a diffuse surface it does not undergo diffuse reflection. Instead, it is simply stored at the intersection point.

In the second pass, the distributed ray tracing algorithm with a MAX_DEPTH of one is used to render the image. The caustics photon map is used to augment the illumination calculations for all rays, but the global photon map is only used for the illumination calculations performed at recursive depth one. In fact, illumination from the global photon map and caustics photon map are all that are used for illumination calculations at recursive depth one. The reason for this is that the global photon map, by definition, represents both direct and indirect illumination of the scene. However, the global photon map is a somewhat rough estimator for the illumination of a scene and so using it to perform illumination calculations at recursive depth zero can introduce unwanted blotchiness in the illumination of the scene. On the other hand, if a sufficiently large number of photons are used in the global photon map, these visual artifacts tend to decrease, and so the global photon map can be used for depth zero illumination calculations in this case.

2.7 Photon Mapping

The algorithms and structures used by the global and caustics photon maps differ only in the photon casting algorithms. In this section, the data structures and algorithms used in creating and using the photon maps are discussed.

2.7.1 Data Representation

Before discussing how to create and use the photon map, it is important to discuss the data representation of a photon, and the structure used to store the photon map. A photon is stored in a 20-byte structure that contains the position, energy, and incident direction of the photon.

When initially constructing the photon map, the photons are stored in a contiguous array. Then, once all photons have been added to the map, the contiguous array is modified to a heap representation of a left-balanced kd-tree. The kd-tree is a spatial sub-division structure. Each internal node of the kd-tree defines an axis-aligned splitting plane that divides the photons into two regions. The leaf nodes of the kd-tree are single photons. By storing the photon map as a heap, eight bytes of storage are saved per photon by not having to store pointers for the children. With a 20-byte photon representation, this amounts to a savings of 40% in storage. To convert the array representation into the kd-tree representation is $O(n \log n)$ complexity [15], and takes only a few seconds to construct for photon maps with n equal to a few million photons.

The kd-tree is a very efficient data structure for the photon map with $O(\log N_p)$ complexity for finding the nearest photon to a given point in a tree with N_p photons, and $O(n_p + \log N_p)$ complexity to find the nearest n_p photons to a given point [15].

Photon Representation

Each photon in the photon map needs to store the position, energy, and incident direction of the photon of light. Furthermore, a 2-bit flag is required by the kd-tree representation to store the axis along which the splitting plane of the node is aligned. This results in a 18.25 byte of storage per photon, which is rounded up to 20 bytes to align to 4 byte boundaries. In C, the structure for a photon is:

```
struct Photon {
  // Photon position
  float x, y, z;
  struct {
    // kd-tree flag
    unsigned short flag:2;
```

```
    // unused space
    unsigned short unused:14;
  };
  // Photon energy; compressed
  RGBE pow[4];
  // Incident direction
  unsigned char theta, phi;
};
```

where x, y, z is the position of the photon, power is the energy of the photon stored as four bytes using Ward's shared-exponent representation [25], and phi and theta is the incident direction of the photon.

The compressed, RGBE, representation of an RGB colour, r, g, b, is calculated as:

```
float v;
int e;
v = max{r, g, b};
v = frexp(v,&e) * 256.0 / v;
power.r = r * v;
power.g = g * v;
power.b = b * v;
power.exp = e + 128;
```

where frexp() is a function from the standard C library that calculates the fractional and integral components of a floating point number.

The incident direction, dx, dy, dz, of the photon is stored, compressed, as a mapping to one of 65536 possible directions as:

$$\text{phi} = \frac{256 * \text{atan2}(dy, dx)}{2 * \pi},$$

$$\text{theta} = \frac{256 * \arccos(dz)}{\pi}.$$

2.7.2 Creating the Photon Map

The photon map is created in a preprocessing step of image synthesis. Before the photon map is created, the user must specify the total number of photons, N_p, that are going to be cast into the scene from the light sources. Once the number of photons to cast into the scene has been defined, the number of photons to cast from each light source must be determined. According to Jensen [15], it is preferable to have a photon map where each photon in the photon map has roughly the same energy.

The total number of photons to cast into the scene and the power of each light source are known. Thus, the number of photons to cast from each light source, such that the energy of photons in the photon map is the same, can be solved for. Let $\mathcal{V}$ be the set of all lights in the scene. Furthermore, let $I_{\mathcal{L}}, A_{\mathcal{L}}, W_{\mathcal{L}}$ be the colour, surface area, and wattage per unit area of light $\mathcal{L} \in \mathcal{V}$, respectively. The colour of the light is defined as an RGB colour, and the wattage is a user-defined scalar that allows for the brightness of the light to be easily controlled. Then, the power of light $\mathcal{L}$ is:

$$\Phi_{\mathcal{L}} = I_{\mathcal{L}} A_{\mathcal{L}} W_{\mathcal{L}}. \tag{2.17}$$

Let $N_{\mathcal{L}}$ be the number of photons cast from light $\mathcal{L}$. Then, the power of a photon cast from light $\mathcal{L}$ will be:

$$\Phi = \frac{\Phi_{\mathcal{L}}}{N_{\mathcal{L}}}. \tag{2.18}$$

Equation 2.18 can be rewritten as:

$$N_{\mathcal{L}} = \frac{\Phi_{\mathcal{L}}}{\Phi}. \tag{2.19}$$

This arrives at an equation that gives the number of photons to cast from light $\mathcal{L}$ to have a photon power of Φ. Using Equation 2.19, a binary search can be used to find how many photons to cast from each light source such that the power of all photons cast is the same and the total number of photons cast is N_p.

max = max$\{\Phi_{\mathcal{L}_i} : \mathcal{L}_i \in \mathcal{V}\}$
min = min$\{\Phi_{\mathcal{L}_i} : \mathcal{L}_i \in \mathcal{V}\}$
while ((max - min) $> \epsilon$)
    Φ = (max + min) $\times 0.5$
    sum = $\sum_{\mathcal{L}_i \in \mathcal{V}} \frac{\Phi_{\mathcal{L}_i}}{\Phi}$
    if (sum == N_p) break
    if (sum $< N_p$)
        min = avg
    else
        max = avg
    endif
endwhile

This algorithm determines the power, Φ, of all photons that will be cast from the light sources. This power and Equation 2.19 are used to determine the number of photons to cast from a given light.

An alternative method to determining how many photons to cast from each light source is to use a probability distribution. The probability of casting a photon from a light source, $\mathcal{L}$, with power $\Phi_{\mathcal{L}}$ is:

$$p = \frac{\Phi_{\mathcal{L}}}{\sum_{\mathcal{L}_i \in \mathcal{V}} \Phi_{\mathcal{L}_i}}.$$

However, the problem with this method is that there is no way of knowing how many photons will be cast from each light source beforehand. Thus, steps must be taken to identify which photons are cast from which light source so that the photon's power may be set properly.

Once the number of photons to cast from each light source is known, all that remains is to cast the photons from the light sources into the scene. The initial position and direction of a photon is determined randomly from the light source from which it is cast. The initial position of a photon is a randomly generated point on the surface of the light from which it is cast; in the case of a point light, the position of the photon is simply the position of the point light. When generating the global photon map, the initial direction of the photon is a randomly generated vector in the half-sphere around the surface normal at the photon's initial position. On the other hand, when generating the caustics photon map, the initial direction is randomly chosen such that the photon will be directed toward an object that generates caustics. The power of the photon is set equal to $\frac{\Phi_{\mathcal{L}}}{N_{\mathcal{L}}}$.

The algorithm used to trace the path of a photon through the scene is very similar to the path tracing algorithm that can be used during image generation. When tracing a photon path, the first intersection of the photon with objects in the scene is found using the same algorithms used to find the first ray-object intersection during ray tracing. Then, using a technique called Russian roulette, it is determined if the photon is to be absorbed into the surface at its intersection point, reflects off of the surface, or refracts through the surface. If the photon intersected a non-mirrored, non-transparent surface point then the photon can either be absorbed at the surface point or diffusely reflected. If the photon is absorbed, it is simply stored in the photon map at the intersection point. Otherwise, it is reflected in a randomly generated direction within the hemisphere on the tangent plane at the intersection point. A probability of absorption, ρ_d, is calculated from the diffuse colour, k_d, at the point of intersection:

$$\rho_d = \frac{k_{d,r} + k_{d,g} + k_{d,b}}{3.0},$$

where $k_{d,r}$, $k_{d,g}$, and $k_{d,b}$ are the red, green, and blue components of k_d. Note that when creating the caustics photon map, the probability of absorption is set equal to 1.0; that is, caustic photons are always absorbed at the first diffuse intersection.

Then, if a uniform distributed random number, $\zeta \in [0, 1]$, is less than ρ_d the photon is absorbed at the intersection point. Otherwise, the photon is diffusely reflected off of the

surface. Determining if a photon should be absorbed in this manner essentially makes a white surface into a photon sink, where all photons are absorbed into it, and makes a black surface into a diffuse mirror, where all photons are reflected from it. Considering that the nearest n_p photons to a point will be used later to estimate the illumination at that point, it makes sense that no photon is stored on a black surface; regardless of the illumination on a black surface, a black surface will always appear black.

Photons are not absorbed on mirrored or transparent surfaces. Thus, if the photon intersects either a mirrored or transparent surface, only whether the photon reflects off of, or transmits through, the surface needs to be determined. Just as in the case of a non-mirrored and non-transparent surface, probabilities for each type of reflection or transmission are calculated based on the material properties at the surface:

$$\rho_d = \frac{k_{d,r} + k_{d,g} + k_{d,b}}{3.0},$$

$$\rho_r = \frac{k_{r,r} + k_{r,g} + k_{r,b}}{3.0},$$

$$\rho_t = \frac{k_{t,r} + k_{t,g} + k_{t,b}}{3.0},$$

where ρ_d, ρ_r, and ρ_t are the probabilities of diffuse reflection, mirror reflection, and transmission, respectively. As above, a uniformly distributed random number, $\zeta \in [0, 1]$, is then generated and used to determine what type of surface interaction occurs:

$\zeta < \frac{\rho_d}{\rho_d+\rho_r+\rho_t}$	$\Rightarrow$	diffusely reflect the photon;
$\zeta < \frac{\rho_d+\rho_r}{\rho_d+\rho_r+\rho_t}$	$\Rightarrow$	mirror reflect the photon;
otherwise	$\Rightarrow$	transmit the photon through the surface.

Regardless of whether the photon is absorbed, reflected, or refracted, it does not lose any power from the interaction with the surface.

2.7.3 Using the Photon Map to Calculate Global Illumination

Once the photon map has been created in the preprocessing step, all that remains is to render the image. When using a photon map, this is typically done using a combination of direct lighting calculations and lighting estimates calculated from the photon map.

Ray Tracing with the Photon Map

The ray tracing algorithm used is Cook's distributed ray tracing algorithm, with a maximum recursive depth of one. Calculating the lighting at a ray's intersection point, however, is done differently based on the current recursive depth. If the recursive depth is zero, then standard direct lighting calculations are used for general illumination calculations, and the

caustics photon map is used to calculate the caustics in the scene. On the other hand, if the recursive depth is greater than zero, no direct lighting calculations are performed; instead, the lighting at the intersection point is calculated solely from the photon map. Note that since a photon will never be absorbed at a mirror or transparent surface, no shading calculations are performed on these surfaces. The modified *trace* function from the distributed ray tracing algorithm follows:

```
function trace(ray, depth)
    object ← firstIntersection(ray)
    x ← intersection point of ray with object
    if (depth = 0)
        colour ← shade(ray, object, x)
        colour + = shade_from_caustics_photon_map(ray, object, x)
    else
        if (object is non-mirrored) AND (object is non-transparent)
            colour ← shade_from_global_photon_map(ray, object, x)
            colour + = shade_from_caustics_photon_map(ray, object, x)
        endif
    endif
    if object is a mirror
        colour += k_r(x) trace(reflected ray, depth)
    endif
    if object is transparent
        colour += k_t(x) trace(refracted ray, depth)
    endif
    if (depth < 1)
        indirect_colour ← black
        for i = 1...N_f do
            R ← random ray from x
            R_colour ← trace(R, depth+1)
            indirect_colour + = addLight(R_colour)
        endfor
        colour + = indirect_colour
    endif
    return colour
```

Finding Photons in the Photon Map

The number of times the photon map is used when rendering a typical scene can be quite staggering. Several million, or even tens or hundreds of millions of lighting estimates are often performed when rendering a single image. Thus, it is important that it is as efficient as possible to calculate lighting estimates through the photon map. Calculating a lighting estimate with the photon map amounts to finding the nearest n_p photons to the point, $\boldsymbol{x}$, at which the illumination is being estimated. Thus, it is essential to find these photons in an efficient manner.

When searching for the nearest n_p photons to the point $\boldsymbol{x}$ in the photon map, all photons within a distance of d_{max} of $\boldsymbol{x}$ are searched. Specifying a maximum search distance allows the search algorithm to prune large portions of the kd-tree, thus speeding up the search. However, occasionally d_{max} might be chosen such that there are fewer than n_p photons within the distance from $\boldsymbol{x}$. In this case, the value of d_{max} is increased by using the number of photons found to estimate the density of photons in this portion of the scene. The value of d_{max} is increased in this manner, until n_p photons have been found or d_{max} has exceeded an absolute maximum search distance. The initial value of d_{max} is typically provided by the user of the system, though it may be possible to calculate an initial search radius by estimating the density of photons in the scene. This algorithm, in pseudocode, follows:

d_{max} = initial search radius
$search_{max}$ = absolute maximum search radius
repeat
  nearest_photons = locate_photons(d_{max})
  if (num_photons_found = 0)
    d_{max}+ = minimum distance increase
  else
    if (num_photons_found < n_p)
      if (d_{max} >= $search_{max}$)
        break
      endif
      $d_{max} = \frac{d_{max} \times n_p}{\text{num_photons_found}}$
      if (d_{max} > $search_{max}$)
        $d_{max} = search_{max}$
      endif
    endif
  endif
until num_photons_found = n_p

If d_{max} is too large, the search through the kd-tree will suffer a performance hit from searching too large a portion of the kd-tree. On the other hand, if d_{max} is too small, then extra searches through the kd-tree will need to be performed to find the photons required to estimate the illumination at $\boldsymbol{x}$.

To search the kd-tree for the nearest n_p photons, a fairly generic nearest neighbour searching algorithm is used that begins at the root node of the tree, and adds photons to a list of nearest photons if they are within distance d_{max} of $\boldsymbol{x}$. During the search, if the number of photons that have been found is less than n_p, then this list is just a standard array. On the other hand, if there are more than n_p photons within the distance d_{max} the list of photons is stored as a max heap. This allows for the furthest photon from $\boldsymbol{x}$ to be very efficiently removed when adding a new photon to the list. Furthermore, when the max heap is full, that is it contains n_p photons, the distance to the furthest photon, in the heap, is used in place of d_{max}. This allows for the search effort to be focused on only the photons in the kd-tree that are candidates for being one of the nearest n_p photons to $\boldsymbol{x}$.

The search for the n_p nearest neighbours of the point $\boldsymbol{x}$ begins at the root of the kd-tree (element 1 in the heap representation that is used for the photon map). If the photon under examination is within d_{max} of $\boldsymbol{x}$, then it is added to a list of nearest photons. d_{max} is then redefined to the largest of the distances from $\boldsymbol{x}$ to the photons in the list if the list contains n_p photons. When the children of the current photon are visited, the one nearest $\boldsymbol{x}$ is always visited first, and the other child is visited only if the distance from $\boldsymbol{x}$ to the splitting plane is less than d_{max}. This algorithm lends itself quite nicely to a recursive format, but an iterative algorithm can be used for efficiency. The iterative algorithm maintains a stack, S, of:

```
struct StackElement {
  unsigned int photon;
  float dist;
}
```

to store the nodes in the kd-tree that still need to be visited during the search. This structure stores the index of a photon in the kd-tree (recall that the photon map is stored as a heapified kd-tree), and an approximation to the distance from $\boldsymbol{x}$ to the photon. The iterative searching algorithm follows.

function locate_photons(d_{max})
    num_photons_found $\leftarrow 0$
    list $\leftarrow$ n_p element array of photons
    list_is_heap $\leftarrow$ false
    $d^2 \leftarrow d_{max} \times d_{max}$

```
S ← empty stack
push node 1 (the root), and distance 0 onto S
while (S is not empty)
    currNode, currNodeDist ← S.pop()
    if (currNodeDist > d^2)
        continue
    endif
    p ← photon currNode
    dist ← ||p->position − x||^2
    if (dist < d^2)
        if (num_photons_found < n_p)
            list[num_photons_found++] ← p
        else
            if (list_is_heap = false)
                convert list into a max heap
                list_is_heap = true
            endif
            remove the photon from the list with maximum distance
            add p to the max heap list
            d^2 ← max{d = ||ph->position − x||^2 : ∀ph ∈ list}
        endif
    endif
    dist ← distance from x to the splitting plane in p
    dist2 ← dist^2
    if (dist > 0)
        if (2×currNode <= N_p) AND (dist2 < d^2)
            push node 2×currNode, and distance dist2 onto S
        endif
        if (2×currNode +1 <= N_p)
            push node 2×currNode+1, and distance 0 onto S
        endif
    else
        if (2×currNode +1 <= N_p) AND (dist2 < d^2)
            push node 2×currNode+1, and distance dist2 onto S
        endif
        if (2×currNode <= N_p)
            push node 2×currNode, and distance 0 onto S
```

endif

endif

endwhile

return list

Calculating the Light Emitted from x via the Photon Map

Once the nearest n_p photons to point x have been found, the emitted light at x, with normal N, in the direction ω is estimated as a sum of the nearest n_p photons:

$$I(x,\omega) \approx \frac{1}{\pi r^2} \sum_{p=1}^{n_p} ((N \cdot \omega_p)\Phi_p), \tag{2.20}$$

where ω_p and Φ_p are the incident direction, and the power of photon p, respectively. r is the maximum distance from x to any of the n_p photons used in the summation.

When the total number of photons in the photon map, N_p, is low, relative to the total surface area of the scene, it is quite likely that the n_p photons used in the irradiance estimate will be quite spread out. In these cases, it is often desirable to filter the irradiance estimate. For this, a cone filter can be used in the sum:

$$I(x,\omega) \approx \frac{1}{\pi r^2 \times R} \sum_{p=1}^{n_p} ((N \cdot \omega_p)(R - r_p)\Phi_p), \tag{2.21}$$

where R is the absolute maximum search distance, and r_p the distance from photon p to the point x. This filter causes photons further away from the point x to have less of an effect on the irradiance at x, while the photons closer to the point x maintain a reasonably high level of effect on the irradiance.

2.8 Motion Blurring

When taking photographs of moving objects with a real camera, they can appear blurred in the photograph. The blurring tends to be very subtle when the object is moving slowly, and can be quite dramatic when the object is moving at high speed. The blur is due to the camera's shutter remaining open for a very short amount of time, exposing the film, to take the picture. If an object in the scene is moving fast relative to the amount of time the shutter remains open, then the light reflected off of, or emanating from, the object is spread over a large area of the film causing the object to appear blurred. Anyone who has ever tried to take a photograph from the window of a moving car, and caught a roadside post in the picture, has seen this phenomenon first hand.

If an image synthesis algorithm, such as ray tracing, wishes to produce truly photo-realistic images, then it needs to be able to duplicate this effect. Fortunately, adding the

ability to reproduce this phenomenon is quite simple for the classical, distributed, and path tracing algorithms.

Adding motion blur support to the three algorithms is done in exactly the same manner. A new constant, N_t, is defined as the number of samples to perform in the temporal domain. Rather than generating only N_s rays from the camera through each pixel in the image $N_s \times N_t$ rays are generated instead. Furthermore, each ray is randomly assigned a time within the interval that the virtual-camera's shutter is open. When casting each ray through the scene, intersection tests are performed with geometry at the position it is located at the random time associated with the ray.

For example, adding motion blur to the distributed ray tracing algorithm yields the following pseudocode:

```
function MotionBlurringDistributedRayTrace
    for each pixel, (i, j), in the output image
        pixelColour ← black
        for k = 1...(N_s × N_t)
            ray ← random ray, from the camera, through pixel (i, j).
            t ← random time in the interval of shutter exposure
            pixelColour += trace(ray, t, 0)
        endfor
        image pixel (i, j) ← pixelColour / (N_s × N_t)
    endfor

function trace(ray, t, depth)
    object ← firstIntersection(ray)
    x ← intersection point of ray with object at time t
    colour ← shade(ray, t, object, x)
    if object is a mirror
        colour += k_r(x) trace(reflected ray, t, depth)
    endif
    if object is transparent
        colour += k_t(x) trace(refracted ray, t, depth)
    endif
    if (depth < MAX_DEPTH)
        indirect_colour ← black
        for i = 1...N_f do
            R ← random ray from x
            R_colour ← trace(R, t, depth+1)
```

```
        indirect_colour += addLight(R_colour)
    endfor
    colour += indirect_colour
endif
return colour
```

2.8.1 Adaptive Motion Blurring

When calculating motion blur as in Section 2.8, $N_s \times N_t$ rays are cast through pixels of the image that do not contain any motion blur. If there were no moving objects, the colour of these pixels would have been calculated with N_s rays to produce a noiseless image. Thus, when no moving objects affect the light incoming to a pixel, $(N_t - 1) \times N_s$ more rays are used to determine the colour of the pixel than are required.

To reduce the amount of over-calculation performed when rendering images containing motion blur, a technique that adaptively increases the number of rays used in regions containing blur can be employed. This method of calculating motion blur casts N_s rays per pixel, and the rays cast are not assigned a random time. Instead, each of these N_s non-temporal rays is assumed to exist over the interval of time that the camera's shutter is open. Whenever it is determined that a non-temporal ray passes through the path of a moving object the ray's path is sampled at N_t temporal samples, and the results are averaged to arrive at a result for the non-temporal ray.

For example, consider the two dimensional scene in Figure 2.9 rendered using the path tracing algorithm. This scene contains two light sources, $\boldsymbol{L1}$ and $\boldsymbol{L2}$, two stationary rectangles, object O1 and O3, and a moving rectangle, object O2. The extents of object O2's movement, during the shutter exposure interval, are denoted by a dashed rectangular outline.

The ray $\boldsymbol{R1}$ is cast from the camera, and intersects at the point $\boldsymbol{a}$. The direct lighting of $\boldsymbol{a}$ from the two light sources is then calculated. In calculating the lighting from $\boldsymbol{L1}$, notice that the path of light from $\boldsymbol{L1}$ to $\boldsymbol{a}$ passes through the path of motion of object O2. Thus, the lighting from $\boldsymbol{L1}$ to $\boldsymbol{a}$ is sampled at N_t temporal samples and then averaged to arrive at the lighting contribution from $\boldsymbol{L1}$. On the other hand, the path of light from $\boldsymbol{L2}$ to $\boldsymbol{a}$ does not pass through the path of a moving object, and so the lighting contribution of $\boldsymbol{L2}$ is calculated as if there were no moving objects in the scene. The path tracing algorithm will then generate a random diffuse reflection of the ray $\boldsymbol{R1}$, $\boldsymbol{R1'}$, to sample the indirect lighting of the point $\boldsymbol{a}$. Notice that $\boldsymbol{R1'}$ passes through the path of motion of object O2. Thus, the adaptive temporal sampling algorithm will sample $\boldsymbol{R1'}$ at N_t uniformly distributed random temporal samples. Some of these samples will yield an intersection at point $\boldsymbol{b}$, and the remainder will yield an intersection at $\boldsymbol{c}$. Each intersection occurs at a different time, and

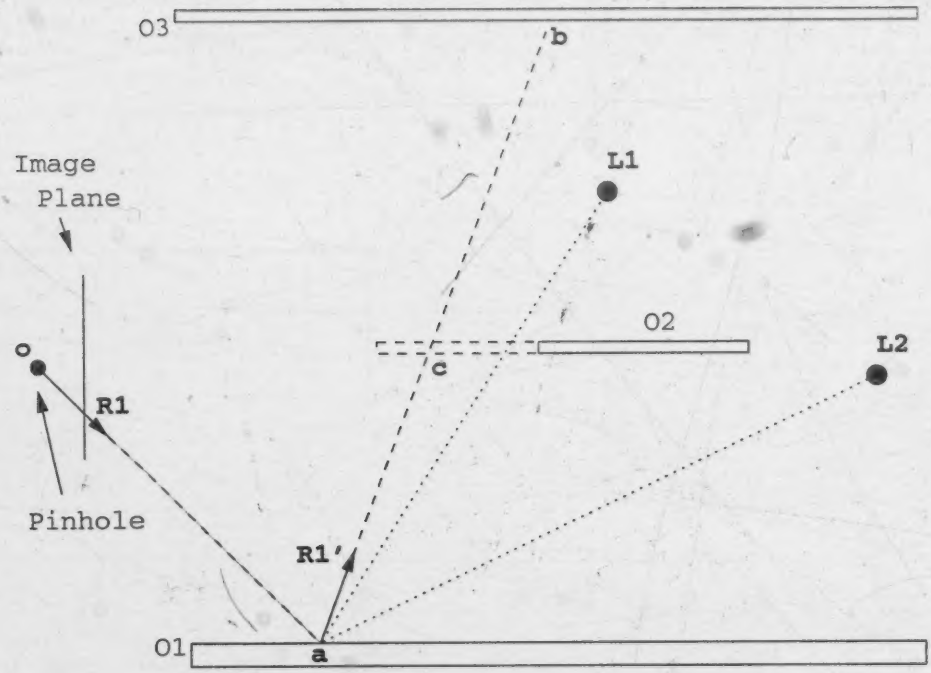

Figure 2.9: Demonstration of the adaptive motion blurring algorithm.

so the lighting of b and c is calculated at different times for each ray. The results of the N_t lighting calculations are then averaged, to yield a light colour for $R1'$.

When applied to the path tracing algorithm, the adaptive motion blurring algorithm is:

```
function PathTraceImage()
    for each pixel
        pixel colour = black
        for sample = 1...N_s
            ray = random ray from the camera
                    through the current pixel
            pixel colour += trace( ray )
        pixel colour = pixel colour / N_s

function trace( ray )
    colour = black
    for depth = 1... MAX_DEPTH
        point = n(R)
        if m(point,ray.origin)
            blur colour = black
```

```
            S = {t_i : i = 1...N_t, t_i = random time}
            for t ∈S
                blur colour += trace( ray, t )
            colour = blur colour / N_t
            return colour
        N = the surface normal at point
        colour += shade( point, N )
        ray = randomly reflected ray from point
    return colour


function shade( point, normal )
    colour = black
    for each light source
        lightPos = random position on the light
        if m(point,lightPos)
            blur colour = black
            for i = 1...N_t
                t = random time
                if v(point,lightPos,t)
                    blur colour += d(point,lightPos)
            colour += blur colour / N_t
        else
            if v(point,lightPos)
                colour += d(point,lightPos)
    return colour
```

where:

N_s	Number of rays cast per pixel.
N_t	Number of temporal samples taken per ray.
$v(x,y)$	True if point y is visible from point x at the start of the frame.
$v(x,y,t)$	True if point y is visible from point x at time t.
$m(x,y)$	True if the line segment $\bar{xy}$ intersects the path of a moving object.
$n(R)$	The nearest intersection point with a

non-moving object along ray R.

$n(R,t)$ The nearest intersection point with any object along ray R at time t.

$d(x,y)$ The direct illumination of point x from point y.

2.9 The Time Dependent Photon Map

The photon map described in Section 2.7, although quite good for static scenes, will not produce an accurate illumination map for scenes containing moving objects. The reason is that as an object moves through a scene, the illumination of the scene will change in response to the moving object; for instance, the shadow a moving object casts will move along with the object. However, the photon map described in Section 2.7 is created at a fixed time, and so illumination that should change as an object moves will remain static.

To remedy this, Cammarano and Jensen [3] extend the standard photon mapping algorithm to handle indirect illumination of motion blurred objects by adding a time field to the photon representation, and making some slight modifications to the photon mapping algorithm. When they initially create a photon they also generate a random time for the photon; the photon is assumed to be travelling through the scene at this random time. The energy of the photon is also scaled by the shutter speed, t_s, so that the energy cast from each light source is no more than the energy it would have been able to cast during the exposure interval. Thus, the energy of a photon cast from light $\mathcal{L}$ with energy $\Phi_{\mathcal{L}}$ will be $\frac{\Phi_{\mathcal{L}} \times t_s}{N_{\mathcal{L}}}$ where $N_{\mathcal{L}}$ is the total number of photons cast from light $\mathcal{L}$. The remainder of the photon tracing portion of the algorithm remains the same.

The next difference comes in the second pass of the photon mapping algorithm. In this pass, there are two types of rays that will be used when ray tracing the scene; a time independent ray, and a time dependent ray. The time dependent ray is used to shade areas of the scene affected by motion blur, while the time independent rays are used to shade the remainder of the scene. Given a time dependent ray, $\boldsymbol{R}$, at time t, with direction $\boldsymbol{\omega}$ intersecting the scene at a point $\boldsymbol{x}$, Cammarano and Jensen calculate the illumination of the point $\boldsymbol{x}$, with surface normal $\boldsymbol{N}$, using the summation of the spatially and temporally nearest n_p photons:

$$I(x,\omega,t) \approx \frac{1}{\pi r^2 \Delta t} \sum_{p=1}^{n_p} ((\boldsymbol{N} \cdot \omega_p)\Phi_p)\,, \tag{2.22}$$

where Δt is the maximum time difference of the photons included in the summation. Cammarano and Jensen also propose a two-pass algorithm for finding the n_p photons used in this summation. Pass one of their two-pass algorithm is to find the spatially nearest $\frac{n_p}{\kappa}$ ($\kappa \in (0,1]$) photons to the intersection point $\boldsymbol{x}$. Pass two is to find the n_p temporally nearest photons to the ray time, t, using a quicksort algorithm with randomized partitions.

Although Cammarano and Jensen suggest using the quicksort algorithm with randomized partitions to find these photons, the list of photons does not actually need to be fully sorted by time to find the n_p temporally nearest ones. The median split algorithm [10] can be used to find the n_p temporally nearest photons from the list of n photons in an expected $O(n)$ running time. The median split algorithm simply semi-sorts the list such that all elements below the n_p^{th} element are less than it, and all elements above the n_p^{th} element are greater than it.

Given a time independent ray $\boldsymbol{R}$ with direction ω intersecting the scene at the point $\boldsymbol{x}$ with surface normal $\boldsymbol{N}$, the illumination at $\boldsymbol{x}$ is calculated as a sum of n_p spatially nearest photons:

$$I(\boldsymbol{x},\omega) \approx \frac{1}{\pi r^2 t_s} \sum_{p=1}^{n_p} ((\boldsymbol{N} \cdot \omega_p)\Phi_p) . \tag{2.23}$$

In this summation, the Δt term from Equation 2.22 is replaced with the shutter speed, t_s, because the illumination of $\boldsymbol{x}$ is being calculated over the entire exposure interval.

As with the standard photon map, each photon in the time dependent photon map stores the photon's position, energy, incident direction, and a 2-bit flag for the kd-tree. Photons in the time dependent photon map also store the time at which the photon was traced through the scene. Thus, each photon in the photon map requires 22.25 bytes of storage, which is rounded up to 24 bytes to align the photons in the array to 4 byte boundaries. The following is a typical C declaration of a time dependent photon:

```
struct TimeDependentPhoton {
  // Photon position
  float x, y, z;
  // Time of the photon
  float time;
  struct {
    // kd-tree flag
    unsigned short flag:2;
    // unused space
    unsigned short unused:14;
  };
  // Photon energy; compressed
  unsigned char pow[4];
  // Incident direction
  unsigned char theta, phi;
};
```

2.10 Optimizing Ray-Object Intersection Calculations

A fundamental part of the ray tracing algorithm is the algorithm to calculate the first intersection of a given ray with the objects in the scene. Since many rays are typically used when rendering a single image, the efficiency of the first intersection algorithm used can have a drastic effect on the efficiency of the ray tracing algorithm. Choosing a good first intersection algorithm over a poor one can mean the difference between an image rendering taking hours versus days.

Many different algorithms have been studied to optimize first intersection calculations [1, 4, 9, 12, 13, 18, 21]. As in the algorithm presented in Section 2.1, one possibility is to just ignore optimizing the first intersection algorithm and simply iterate through every object in the scene, calculating the intersection with each, to determine which object the ray will intersect first. This might even work for scenes containing only a handful of simple objects, such as spheres and triangles. But, when the scenes being rendered start to contain more than a handful of objects, or when they start to contain objects with complex ray-intersection formulae, then faster first intersection algorithms are required.

There are two main types of ray-object intersection optimization. The first is the bounding volume method. This method calculates a simple bounding volume around each object that is faster, and easier, to test for intersection against than the object it is bounding. The idea being that if a complex intersection test can be skipped by performing a faster one first, then the time required to find the first ray-object intersection will be reduced.

The other methods are the spatial partitioning methods. In these methods, the scene space is partitioned, using some structure, to allow for rapid pruning of entire regions of a scene.

2.10.1 Bounding Volumes

One avenue of optimization is to use bounding volumes, such as spheres or boxes, to speed up ray-object intersection tests. In this optimization, a bounding volume for each object that totally encloses the object is calculated. Hopefully, the bounding volume will enclose the object as tightly as possible. Then, when performing a ray-object intersection calculation an intersection test with the bounding volume is performed first. If the ray does not intersect the bounding volume, then the ray cannot possibly intersect the object, and so no intersection test with the object is performed. On the other hand, if the ray does intersect the bounding volume, then the ray may intersect the object, and so a regular intersection calculation on the object is performed.

When choosing the type of bounding volume to use, it is important that the time required to calculate a ray intersection with the bounding volume be faster than the time required to calculate a ray intersection with the bounded object. This is important because the speedup

gained by utilizing bounding volumes comes from the speed of calculating the intersection with the bounding volume compared to the speed of calculating the intersection with the bounded object. If it is slower to calculate the intersection with the bounding volume than to calculate the intersection with the bounded object, then utilizing bounding volumes would only serve to slow down the ray tracing algorithm.

Furthermore, the percentage volume of the bounding volume that is not occupied by the object also affects the amount of speedup that can be expected from such methods. If most of the bounding volume is occupied by the object, then when a ray is determined to intersect the bounding volume there is a higher probability that the ray will intersect the object than if more of the volume of the bounding volume was not occupied by the object.

One of the main benefits of using bounding volumes in a ray tracer is that the method can be combined with any other ray intersection optimizations included. Since this particular optimization is local to each object, it is used as a final pruning prior to calculating the intersection between the ray and the object.

Axis Aligned Bounding Box

The axis aligned bounding box is one of the easier to implement bounding volume techniques. An axis aligned box is simply a box that has each edge parallel to one of the three principal axes. To construct an axis aligned bounding box, only the minimum and maximum values of the object along each axis needs to be determined; these values define the axis aligned box that will become the bounding volume.

To quickly determine if a given ray intersects an axis aligned bounding box, Schroeder [22] has devised a quick intersection test based on the Cohen-Sutherland line clipping algorithm [8]. To use this algorithm the ray is converted into a line segment, that will be, essentially, clipped against the three dimensional box. The ray origin is used as one end of the line segment, and either the nearest intersection point found so far or a point that is sufficiently far along the ray so as to pass all objects in the scene is used as the second endpoint of the line segment.

Let $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ be the two endpoints of the line segment being clipped against the axis aligned bounding box. Furthermore, let the extents of the axis aligned bounding box be $\boldsymbol{min} =< x_{min}, y_{min}, z_{min} >$ and $\boldsymbol{max} =< x_{max}, y_{max}, z_{max} >$. The algorithm begins by encoding the position of each point, relative to the box, into a six bit vector. The algorithm to calculate this encoding can be seen in Appendix B.

Using this encoding, a code of zero indicates that the point is inside the bounding box, and thus the line segment definitely intersects the bounding volume, and a non-zero code indicates the point is outside the bounding box. Furthermore, if the bitwise and of two codes is non-zero then the two points are on the same side of the bounding box; thus the

ray cannot possibly intersect the bounding box. These two observations yield two quick tests to either accept or reject the ray for intersection with the bounding box. If the encodings of the two endpoints do not adhere to either of these conditions, then the codes can be used to determine which sides of the box should be tested for intersection with. For example, if the first code contains neither CLIP_RIGHT nor CLIP_LEFT, then there is no need to test for intersection with the $x = x_{max}$ or $x = x_{min}$ planes. These observations yield the following line-segment intersection test:

```
// Returns true if the line segment P1-P2 intersects the
// bounding box defined by min and max
function linesegIntersect(P1, P2, min, max)
    outcode1 ← calculate_outcode(P1)
    outcode2 ← calculate_outcode(P2)
    if (outcode1 == 0) OR (outcode2 == 0) return true
    if ((outcode1 bitAND outcode2) != 0) return true
    if( outcode1 bitAND (CLIP_RIGHT bitOR CLIP_LEFT) )
        if( outcode1 bitAND CLIP_RIGHT ) intercept.x = x_max
        else intercept.x = x_min
        x1 = P2.x - P1.x
        x2 = intercept.x - P1.x
        t = x2/x1
        intercept.y = P1.y + t * (P2.y - P1.y)
        intercept.z = P1.z + t * (P2.z - P1.z)
        if( (intercept.y <= y_max) AND (intercept.y >= y_min)
            AND (intercept.z <= z_max) AND (intercept.z >= z_min) )
                return true
    endif
    if( outcode1 bitAND (CLIP_TOP bitOR CLIP_BOTTOM) )
        if( outcode1 bitAND CLIP_TOP ) intercept.y = y_max
        else intercept.y = y_min
        y1 = P2.y - P1.y
        y2 = intercept.y - P1.y
        t = y2/y1
        intercept.x = P1.x + t * (P2.x - P1.x)
        intercept.z = P1.z + t * (P2.z - P1.z)
        if( (intercept.x <= x_max) AND (intercept.x >= x_min)
            AND (intercept.z <= z_max) AND (intercept.z >= z_min) )
                return true
```

endif
if(outcode1 bitAND (CLIP_FRONT bitOR CLIP_BACK))
if(outcode1 bitAND CLIP_FRONT) $\boldsymbol{intercept}.z = z_{max}$
else $\boldsymbol{intercept}.z = z_{min}$
z1 = $\boldsymbol{P_2}$.z - $\boldsymbol{P_1}$.z
z2 = $\boldsymbol{intercept}$.z - $\boldsymbol{P_1}$.z
t = $\frac{z2}{z1}$
$\boldsymbol{intercept}$.x = $\boldsymbol{P_1}$.x + t * ($\boldsymbol{P_2}$.x - $\boldsymbol{P_1}$.x)
$\boldsymbol{intercept}$.y = $\boldsymbol{P_1}$.y + t * ($\boldsymbol{P_2}$.y - $\boldsymbol{P_1}$.y)
if(($\boldsymbol{intercept}$.x <= x_{max}) AND ($\boldsymbol{intercept}$.x >= x_{min})
AND ($\boldsymbol{intercept}$.y <= y_{max}) AND ($\boldsymbol{intercept}$.y >= y_{min}))
return true
endif
return false

2.10.2 Spatial Partitioning

Spatial partitioning algorithms are essential to optimizing the first ray-intersection calculation when more than a handful of objects are in the scene being rendered. In fact, spatial partitioning algorithms can improve on the rendering time of a scene containing as little as two objects. These algorithms partition the scene space to allow large portions of the space to be pruned with little calculation. Each different spatial partitioning algorithm functions better under different circumstances; thus, care must be taken when choosing which spatial partitioning method to utilize in a ray tracer.

Voxel Grids

One such spatial partitioning algorithm is to partition the scene space using a voxel grid [1, 18]. To construct a voxel grid, an axis aligned bounding box for the entire scene is calculated, which is then divided uniformly into rectangular voxels. Each voxel in the partition is then assigned a list of the scene objects that intersect the voxel; if there are any. Once the grid has been created, an algorithm based on Bresenham's line drawing algorithm can be used to traverse the grid, searching for the nearest intersection point of a given ray. This method avoids ray-object intersection calculations on any object that does not intersect a voxel in the path of the ray.

Unfortunately, the uniform voxel grid assumes that objects are fairly uniformly distributed in the scene. The performance of the optimization can potentially be compromised if objects are dense in some portions of the scene, and sparse in the remainder. Since the grid parameters are preset manually, it is incapable of adapting itself to the layout of each

scene. An incorrect choice of grid parameters, by the user of the system, will degrade performance of the optimization. Consider, for example, the following scene. Imagine a very detailed model of a single floor of an office building. The user, an architect, wishes to use ray tracing to create a fly-through animation of the office space so he can demonstrate the layout to his clients. Each office in the model contains a desk, chair, various office supplies on the desk, and many other objects that would be expected in an office. Since the architect is unsure of the effect of the parameters for the voxel grid optimization, he decides to set the voxels in the grid to be five foot cubes; just large enough to completely enclose an office desk, and all objects on the desk, within a single voxel.

Any time, while rendering the fly-through of the office floor model, that this desk appears in the rendered image, rendering the portion of the image containing the desk will be quite slow due to the poor choice of voxel size. This is because the desk and the multitude of items on the desk are all contained in the same voxel. Thus, any time a ray intersects the desk's voxel, there will be tests for intersection with each of the multitude of objects in the voxel; even if the ray ends up intersecting the base of a leg of the desk, intersection tests with the objects on top of the desk will still be performed.

Lastly, sparse scenes will cause a large number of voxels to remain empty of objects, which will not only waste memory but will also slow down traversal through the grid; even though a voxel may be empty, the voxel traversal algorithm will still need to visit the voxel.

Octrees

Alternately, the scene space can be partitioned by using an octree [4, 21]. An octree is a type of hierarchical spatial subdivision where each non-leaf node in the tree is divided into eight sub-nodes. Each leaf of the tree contains a list of the objects wholly, or partially, enclosed by the axis aligned bounding box for the leaf.

Construction of an octree begins with an axis aligned bounding box for the entire scene. The bounding box is then recursively subdivided into eight sub-boxes until a subdivision contains fewer than some number, m, of scene objects. At this point, a leaf node is created with a list of objects that are at least partially enclosed by the axis aligned bounding box of the leaf.

Once the octree has been created, it can be used to prune potentially large portions of the scene space while determining the first intersection of a ray among the scene objects. Similar to the voxel grid approach, ray intersection calculations are performed only on the objects in the lists of the leaf nodes that are intersected by the ray. Chang [4] discusses both recursive and non-recursive methods for traversing an octree while performing ray-intersection calculations.

Although creating an octree from a list of objects is a reasonably simple task there are

no simple ray traversal algorithms for the structure [4]. Fortunately, the octree is capable of adapting, albeit in a limited capacity, to the layout of objects within the scene. However, the amount the structure is able to adapt is restricted by the requirement to create eight subdivisions whenever a node is split.

KD-Trees

Finally, the scene space can be partitioned using a kd-tree [4, 9, 12, 13]. A kd-tree, like an octree, is a hierarchical subdivision structure. However, a kd-tree differs from an octree in that it only subdivides each node into two subnodes. The leaf nodes of a kd-tree are identical to those of an octree, in that they contain a list of the scene objects that are at least partially enclosed by the axis aligned bounding box for the leaf.

Construction of a kd-tree begins in the same manner as constructing an octree; that is, it begins with an axis aligned bounding box that encloses the entire scene. A node split is done by choosing an axis aligned splitting plane that evenly, or as evenly as possible, divides the list of objects in the node in two; those to the left and those to the right of the splitting plane. Objects that straddle the splitting plane are added to both lists of objects. The node is then split along the splitting plane into two children nodes. Constructing and traversing kd-trees has been extensively studied by [12].

Unlike an octree, traversing a kd-tree to determine ray-intersections is moderately simple. Traversal begins at the root node of the tree, and proceeds to recursively visit the subnodes of each node according to the orientation of the ray relative to the splitting plane of the node. Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be the two intersection points of the ray with the axis aligned bounding box for the current node. The children of the node are pruned based on the position of $\boldsymbol{A}$ and $\boldsymbol{B}$ relative to the the position of the splitting plane for the node. For instance, if $\boldsymbol{A}$ and $\boldsymbol{B}$ are both on, say, the left side of the splitting plane, then there is no chance that the ray will intersect with any object on the right side of the splitting plane. Thus, the right child of the current node would be pruned from the search. On the other hand, if $\boldsymbol{A}$ and $\boldsymbol{B}$ are on opposite sides of the splitting plane, then the ray may intersect objects on either side of the splitting plane. Thus, no child nodes are candidates for pruning in this case. Table 2.2 summarizes how to prune children during traversal of the kd-tree based on the locations of $\boldsymbol{A}$ and $\boldsymbol{B}$ relative to the splitting plane.

Position of $\boldsymbol{A}$	Position of $\boldsymbol{B}$	Child, or children, to prune
Left	Left	Right
Left	Right	None
Right	Right	Left
Right	Left	None

Table 2.2: Child nodes to prune during kd-tree ray traversal.

By alternating the axis along which the splitting plane is chosen, at each level of the

kd-tree, the structure gains in adaptability over and above the octree. Furthermore, the kd-tree construction and traversal algorithms are relatively simple to develop. As shown by Havran [12], kd-tree traversal outperforms other ray-intersection optimization algorithms in a large variety of test scenes.

2.11 Efficient Shadow Calculations

As discussed in Section 2.3.1, before a point light can be included in the lighting calculation for a given point, $\boldsymbol{x}$, it must be determined whether the light source is visible from $\boldsymbol{x}$. To perform this visibility test Section 2.3.1 discusses utilizing the first ray-intersection test; if there is a first intersection between the light at $\boldsymbol{x}$, then the light is not visible. However, using the first intersection test to determine visibility is grossly inefficient. When determining if a point is visible to another point, it does not matter which object is the first to block visibility; it only matters that there is an object blocking visibility.

To speed up visibility calculations, a new visibility test is based on the first ray-intersection test. This visibility test is done in much the same manner as the first intersection calculation described in Section 2.3.1 with one subtle difference; only a single object intersection needs to be found for us to determine that $\boldsymbol{L}$ is not visible from $\boldsymbol{x}$. This is a very important observation, as it allows the intersection calculation routine to be aborted as soon as an intersection is found. This speeds up the ray tracing by reducing the number of ray-intersection calculations performed.

2.12 Precalculating Irradiance

When the number of photons in the photon map is a fraction of the total number of irradiance estimates that will be calculated through the photon map, an optimization presented by Christensen [6] can be applied to gain a very large speed up when generating an image. This optimization precalculates the irradiance at each of the photon locations, and stores this irradiance back in the photon. Then, during rendering when the irradiance at a point $\boldsymbol{x}$ is required, the photon map is searched for the nearest photon to $\boldsymbol{x}$ and the irradiance that has been stored in that photon is used as the irradiance at $\boldsymbol{x}$. This will introduce some error to the calculated irradiance at $\boldsymbol{x}$; in fact the irradiance of the scene forms a Voronoi diagram. Given that this optimization is reported [6] to allow for a seven to ten times speed-up in the total time to render an image, this error is acceptable.

When calculating the irradiance at a point $\boldsymbol{x}$, the surface normal at $\boldsymbol{x}$ is required. Thus, in order to calculate the irradiance at a photon location the surface normal at the point the photon was absorbed will be required. The most obvious place to store the surface normal is with each photon. To save space, the surface normal is stored as two bytes in exactly the

same way as the incident direction of the photon is stored. Additionally, the precalculated irradiance is stored within each photon. Doing so adds another four bytes to the photon representation. In total, another six bytes is added to the photon representation for this optimization, bringing the total photon size to 24.25 bytes, which is rounded up to 26 bytes for alignment:

```
struct Photon {
  // Photon position
  float x, y, z;
  struct {
    // kd-tree flag
    unsigned short flag:2;
    // unused space
    unsigned short unused:14;
  };
  // Photon energy; compressed
  RGBE pow[4];
  // Incident direction
  unsigned char theta, phi;
  // Surface normal at the photon position
  char normal_phi, normal_theta;
  RGBE precalculated_irradiance;
};
```

2.12.1 Precalculating Irradiance in the Time Dependent Photon Map

Christensen's irradiance precalculation method can also be applied to Cammarano and Jensen's time dependent photon map. Unlike ray tracing with the standard photon map, when ray tracing images with the adaptive motion blurring algorithm, both time dependent irradiance calculations and time independent irradiance calculations are used. Fortunately, each photon in the time dependent photon map has a time associated with it. So, this information can be utilized when precalculating the irradiance of the various photon locations.

Each time dependent photon stores two different precalculated irradiances; the time independent precalculated irradiance, and the precalculated irradiance at the time stored in the photon. Thus, the size of each time dependent photon is increased by ten bytes to accommodate the extra data; four bytes for each of the precalculated irradiances, and two bytes for the surface normal. This yields the following time dependent photon representation when using Christensen's method:

```
struct TimeDependentPhoton {
  // Photon position
  float x, y, z;
  // Time of the photon
  float time;
  struct {
    // kd-tree flag
    unsigned short flag:2;
    // unused space
    unsigned short unused:14;
  };
  // Photon energy; compressed
  unsigned char pow[4];
  // Incident direction
  unsigned char theta, phi;
  // Surface normal at the photon position
  char normal_phi, normal_theta;
  // The time independent precalculated irradiance
  RGBE precalc_irrad;
  // The time dependent precalculated irradiance.
  RGBE precalc_irrad_time;
};
```

During the irradiance precalculation stage of the algorithm, the time independent irradiance at each photon location is calculated and stored in the "precalc_irrad" field of the photon. Furthermore, the time dependent irradiance is calculated at the time and location of each photon and stored in the "precalc_irrad_time" field of the photon.

When ray tracing an image, the precalculated irradiance used depends on the type of irradiance required (either time dependent or time independent irradiance). Calculating time independent irradiance is done by finding the spatially nearest photon, and using the time independent irradiance stored in the photon. Calculating time dependent irradiance at point x at time t is done by finding the photon such that the following distance metric is minimized:

$$||x - x_p||^2 + \kappa(t - t_p)^2, \tag{2.24}$$

where $\kappa > 0$ is a constant controlling the weight that the time difference has on the metric, and x_p and t_p are the position and time of the photon, respectively. The time dependent irradiance stored in the photon found is used as the irradiance at x and time t.

Chapter 3

Efficient Algorithms

Ray tracing algorithms utilize many different algorithms many times during the course of ray tracing a single image. These algorithms are typically used millions, if not tens or hundreds of millions, of times while ray tracing a single scene. This chapter discusses several new improvements to four of these algorithms.

First is an improvement to the adaptive motion blurring algorithm presented in Section 2.8.1. This adaptive temporal sampling technique causes more sample points to be used in the temporally sampled regions of the image than in the non-temporally sampled regions of the image. When an object is moving at a higher velocity with respect to the frame exposure delay, a potentially large percentage of the temporal samples will strike the stationary objects behind the moving objects. Thus, the number of point samples taken in regions of the image that have been supersampled in the temporal domain could be substantially larger than the number of point samples taken in other regions. This means that the rendering equation will be evaluated to a higher accuracy for stationary objects in regions with motion blur than in regions of the image that are not motion blurred. The result is that stationary objects in motion blurred regions of the image can have a lower variance than the same stationary object would have if it was not in a motion blurred region of the image. The improvement to the adaptive motion blurring algorithm presented removes this variance discrepancy while speeding up the adaptive motion blurring algorithm.

Second is an improvement to Christensen's irradiance precalculation method for photon maps presented in Section 2.12. When using Christensen's method it is possible that some of the calculated irradiance values will not be used when ray tracing the scene. In fact, unless the photon map is used to calculate more irradiance values than the number of irradiance values that were precalculated, Christensen's method will not offer any speedup, and will likely slow down the ray tracer. The improvement to Christensen's irradiance precalculation method presented in this chapter allows the photon map to function as an irradiance cache without adding any more storage than required by Christensen's method. Furthermore, caching irradiance values rather than precalculating them allows for the number of irradiance

values calculated to scale better when a low number of irradiance values are required during ray tracing.

Third is an improvement to the time dependent photon map. The time dependent photon map does not adapt, at all, to the properties of motion of the scene being ray traced. In fact, the number of photons used is solely determined by the user of the ray tracer. The improvement presented in this chapter combines a time independent and time dependent photon map into an adaptive photon map that will adapt to the motion properties of the scene. Furthermore, this adaptive photon map reduces the memory requirements of the photon map and is capable of outperforming the time dependent photon map.

Last is an improvement to the adaptive photon map presented in this chapter. This improvement adds a new type of photon, the interval photon, to the photon map. Adding this new photon reduces the memory requirements of the adaptive photon map by storing, in a single photon, a number of time dependent photons. Furthermore, adding this new photon type is capable of improving the running time of the ray tracer.

3.1 An Oversampling Problem with the Adaptive Motion Blur Algorithm

The adaptive motion blurring technique that has been discussed in Section 2.8.1 may seem to be a very good improvement on the surface. Unfortunately, there is a discrepancy in the accuracy of the colour for rays that are sampled in the temporal domain and the colour for rays that are not. This discrepancy is discussed in the framework of the path tracing algorithm, though it exists in other ray tracing algorithms as well.

Consider a single ray, $\boldsymbol{R}$, that passes through the path of motion of one moving object, O_m, before hitting a stationary object, O_s, at the point $\boldsymbol{P}$. Since the ray passes through the path of O_m, the adaptive temporal sampling portion of the algorithm samples $\boldsymbol{R}$ in the temporal domain so that the image of O_m is blurred. Let us assume that $\boldsymbol{R}$ does not intersect O_m at ρ percent of the temporal samples generated. If there are N_t temporal samples, then $\rho \times N_t$ of the temporal samples perform an illumination calculation at the same point on O_s, but at different times. Furthermore, $\rho \times N_t$ different diffuse reflections are computed from the same point on O_s. Thus, the illumination at point $\boldsymbol{P}$ is calculated as the average of $\rho \times N_t$ potentially different colours due to the illumination from the different diffuse reflections.

Now, consider the same scene and the same ray, $\boldsymbol{R}$, but remove the object O_m. This time, since $\boldsymbol{R}$ does not pass through the path of a moving object, prior to intersecting O_s at $\boldsymbol{P}$, the adaptive temporal sampling algorithm does not sample $\boldsymbol{R}$ in the temporal domain. Thus, the algorithm performs a single illumination calculation at $\boldsymbol{P}$ plus a single diffuse reflection. Thus, the light colour of ray $\boldsymbol{R}$ is calculated using a single sample point; unlike

the previous case when the object O_m was in the scene.

There is an obvious discrepancy in the accuracy with which point $\boldsymbol{P}$ is shaded in these two instances. In the first instance, whenever $\rho \times N_t > 1$ the colour of $\boldsymbol{P}$ is calculated with more samples than in the second instance. Thus, we can expect that the shading of $\boldsymbol{P}$ in the first instance contains a better approximation of the global illumination at $\boldsymbol{P}$.

3.1.1 Variance Invariant Adaptive Motion Blurring Algorithm

The problem with this adaptive motion blurring technique is that it calculates the illuminance of stationary objects using more sample points within motion blurred regions than within the non-blurred regions of the image. This variation in the number of sample points creates a discontinuity in the variance of the colour of stationary objects between blurred and non-blurred regions of the image. In this section, a new variance invariant adaptive motion blurring algorithm (VIAMB) is presented that can alleviate the problem mentioned above.

Figure 3.1 demonstrates this discontinuity. The area of the image where rays were sampled in the temporal domain can be determined visually. There is a distinct pill-shaped region of the image in which the red background has less variance than the remainder of the image. In fact, close visual inspection, at multiple levels of magnification, is unable to notice any noise in the pill-shaped region. In contrast, the red portion of the image that is not within this smooth pill-shaped region has a large variance.

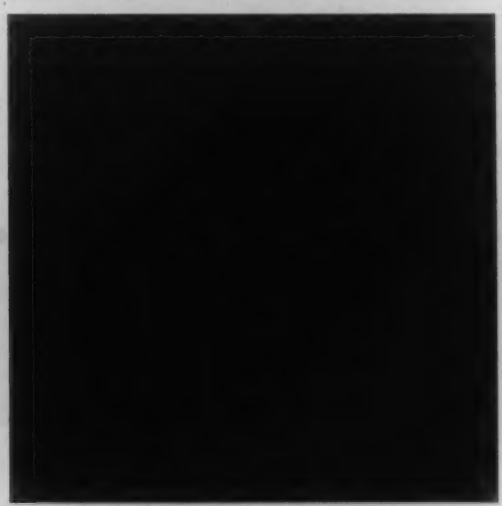

Figure 3.1: A fast moving blue sphere ray traced using the adaptive motion blurring algorithm of Section 2.8.1 with $N_s = 10$ and $N_t = 100$.

The solution to this problem should strive to reduce the number of sample points used in illumination calculations of stationary objects within regions of motion blur, while maintain-

ing the number of sample points used when calculating the illumination of moving objects.

The new method combines all temporal samples that strike a stationary object into a single sample point. Furthermore, the illumination of this sample is weighted such that it has the same contribution as the temporal samples that were combined.

The new algorithm maintains a set, S, of temporal samples. When this set is empty, the ray currently being traced has not yet passed through the path of a moving object, and thus has not been sampled in the temporal domain. When the current ray passes through the path of a moving object, the set S is set to contain the times at which the ray, $\boldsymbol{R}$, is sampled in the temporal domain. For each temporal sample, t, in S, if the current ray strikes a moving object at time t, then t is removed from S and the illumination of the ray at time t is calculated as usual; that is, S only contains the temporal samples at which the current ray does not strike a moving object. Lastly, the illumination of the stationary object struck is weighed by $\frac{|S|}{N_t}$.

This method is able to prevent calculating the illumination of stationary object, which are within motion blurred regions, with more samples than they would be shaded with in regions that are not motion blurred. Furthermore, objects in the scene that should be motion blurred (namely, moving objects) are not shaded with any fewer, or greater, sample points than with the algorithm in Section 2.8.1.

3.1.2 Results

This section contains the image quality and timing results obtained using the adaptive motion blurring algorithm of Section 2.8.1 and the proposed algorithm above.

In Figures 3.1 and 3.2 the results of the original adaptive motion blurring algorithm and the modifications proposed in this thesis are displayed, respectively. Both of the images show a fast moving blue sphere moving from the lower left to the upper right in front of a stationary red plane. Also, both images were rendered with $N_s = 10$ and $N_t = 100$. It should be noted that such extreme values for N_s and N_t were chosen for these images to make the artifacts easier to see on paper. The artifacts are still noticeable for values of N_t as low as N_s.

Figure 3.2 shows the same image found in Figure 3.1 rendered using the new method presented in this thesis. The red background within the motion blurred portion of the image is no longer as smoothly shaded as it is in Figure 3.1. It is noteworthy that there is no visible difference between the red background that is within the motion blurred portion and the red background that is not within the motion blurred portion. Furthermore, the amount of blue motion blur between the two images looks the same.

The data in Figure 3.4 and Table 3.1 were acquired by ray tracing a scene similar to the scene in Figures 3.1 and 3.2. In this test scene, the blue sphere was replaced with a

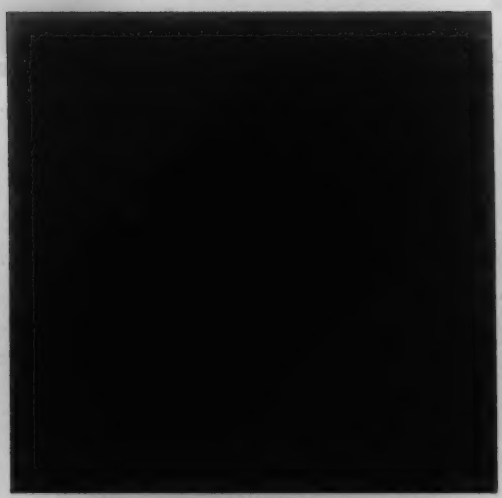

Figure 3.2: A fast moving sphere ray traced using the VIAMB algorithm proposed in this thesis with $N_s = 10$ and $N_t = 100$.

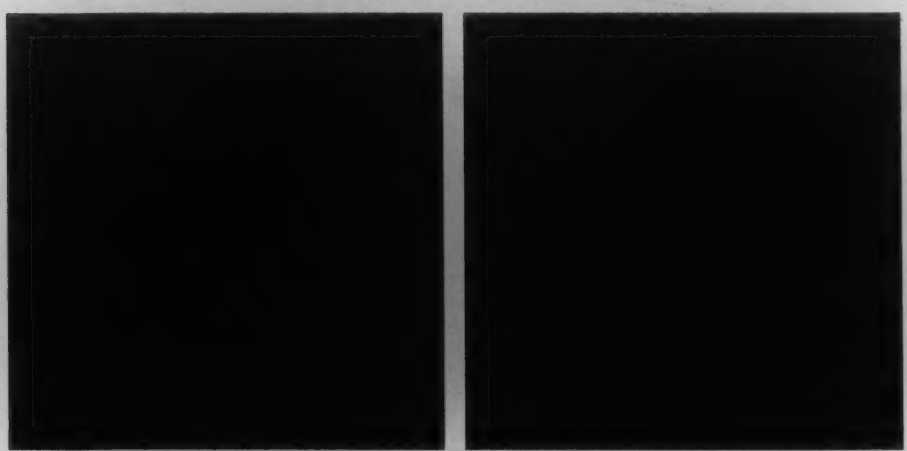

Figure 3.3: Side-by-side comparison of the images in Figure 3.1 (left) and Figure 3.2 (right).

transparent plane to allow a more accurate variance calculation by reducing interference caused by the blue sphere.

Figure 3.4 shows the variation of the variance of pixels in a region inside and a region outside of the motion blurred area of the image with respect to N_t. The variance between the two regions using the VIAMB algorithm is similar, and the variance does not change very dramatically as N_t increases. On the other hand, the variance between the two regions for the existing adaptive motion blurring algorithm differ dramatically as N_t increases.

Table 3.1 shows the time, in minutes, taken to generate the images used in Figure 3.4.

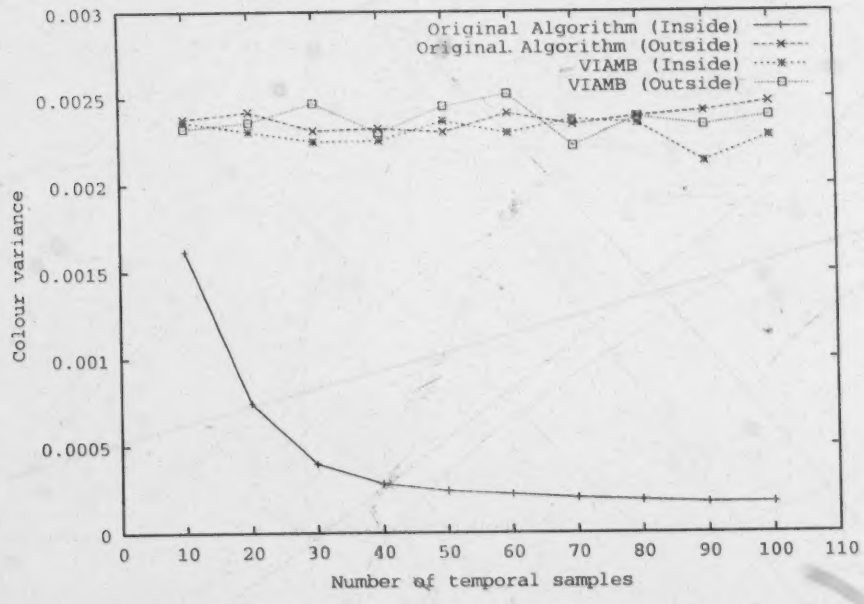

Figure 3.4: Variation of the variance of the red plane both inside and outside of the motion blurred region as N_t increases.

The time taken to generate these images using the VIAMB algorithm is approximately half of the time taken to generate the images using the original algorithm. This difference in running time is primarily due to the reduced number of illumination calculations performed. It should be noted that the time saved when using the VIAMB algorithm is proportional to the portion of the image that is affected by motion blur. Thus, in images with relatively little motion blur this difference will be smaller.

N_t	**Original Algorithm**	**VIAMB**
10	6.28	4.01
20	12.48	7.09
30	18.82	10.12
40	26.03	13.15
50	31.51	16.14
60	36.68	19.27
70	42.67	22.46
80	49.00	25.61
90	54.92	28.19
100	61.89	31.30

Table 3.1: Time (in minutes) taken to ray trace the scene used in Figure 3.4 with $N_s = 10$ on a 2.26 GHz Pentium 4 processor with 512Mb of memory running Linux kernel version 2.4.9.

To better understand the effects of the VIAMB algorithm on the speed of rendering an image, a test suite was composed. Each scene in the test suite is of a $40 \times 40 \times 40$ enclosed

box, with a 10×10 rectangular area light at the top, and containing a variable number of moving diffuse spheres. The moving spheres in each scene are generated such that scene one contains one sphere, and scene $i + 1$ contains the same spheres as scene i plus one additional randomly generated sphere. The spheres are generated such that no two spheres will intersect each other at any point along their motion, nor will any sphere intersect the walls of the box. Furthermore, the velocity of each sphere is randomly generated, as is the radius. The radius is generated in the range $[3, 5]$, and each component of the velocity is generated in the range $[-5, 5]$.

Scene #	Original Algorithm	VIAMB	% Speedup
1	386	339	13.8%
2	485	421	15.2%
3	657	550	19.5%
4	1021	882	15.8%
5	1370	1276	7.4%
6	1445	1340	7.8%
7	1622	1491	8.8%
8	1890	1748	8.1%
9	1883	1740	8.2%
10	2011	1872	7.4%
11	2081	1938	7.4%
12	2085	1948	7.0%
13	2274	2109	7.8%
14	2324	2158	7.7%
15	2467	2309	6.8%
16	2503	2360	6.1%
17	2522	2378	6.1%
18	2746	2574	6.7%
19	2823	2645	6.7%
20	2861	2701	6.0%

Table 3.2: Time (in seconds) taken to ray trace the twenty moving spheres scenes in the test suite on a 2.26 GHz Pentium 4 processor with 512Mb of memory running Linux kernel version 2.4.9.

Table 3.2 presents the results of running both the adaptive motion blurring algorithm of Section 2.8.1 (the "Original Algorithm") and the VIAMB algorithm. The results were generated by rendering a single 300×300 pixel image using the path tracing algorithm with $N_s = 50$, MAX_DEPTH $= 3$, and $N_t = 10$. In this test suite, the modified adaptive motion blurring algorithm yields a speedup, over the original adaptive motion blurring algorithm, of between 6% and 19.5%. The speedup starts quite high, but as more spheres are added the speedup starts to drop slowly. The large drop in the speedup between scene number four and scene number five corresponds to the addition of an extremely slow moving sphere in the center of the image; until this point, all of the spheres in the scene were moving at quite high speeds.

Figure 3.5 shows a side-by-side comparison of scene nine from the test suite, as generated

Figure 3.5: Scene nine from the test suite ray traced using both adaptive motion blurring algorithms. The left image was produced using the original algorithm, and the right was produced using VIAMB.

by both adaptive motion blurring algorithms. See Appendix C for more example images from this test suite.

3.2 On Demand Irradiance Caching with Photon Maps

The irradiance precalculation method for photon maps proposed by Christensen [6] blindly calculates the irradiance at all photon locations, regardless of whether or not a given calculation will actually be used while rendering the image. If a large number of these precalculated irradiance values are not used while rendering the image, then performance of the ray tracer will suffer. In this section, a modification to Christensen's optimization, called on demand irradiance caching, is presented that eliminates calculating irradiance values that are not used to render the image.

Rather than precalculating irradiance at every photon location, irradiance is calculated only as it is needed. To do this, a flag is added to each photon that records whether or not the irradiance has been calculated at the photon location. Fortunately, since there are 14 bits of unused space in each photon, adding this flag does not increase the size of a photon.

During rendering, the irradiance at the point $\boldsymbol{x}$ is calculated by finding the nearest photon in exactly the same manner as in Christensen's method. The flag is then checked, and if it indicates that the irradiance at the photon's location has not yet been calculated then the irradiance is calculated at the photon's location. The calculated irradiance is then stored in the photon, and the flag is set to indicate that the irradiance has been calculated. The calculated irradiance stored in the photon is then used as the irradiance at $\boldsymbol{x}$.

In the case of time dependent photons, two flags are needed rather than one. One flag records whether or not the time independent irradiance has been calculated and stored at

the photon location, and the other records whether the time dependent irradiance has been calculated and stored at the photon location.

3.2.1 Results

This section contains a comparison of results obtained using both Christensen's irradiance precalculation optimization, and the proposed on demand caching optimization with the time dependent photon map.

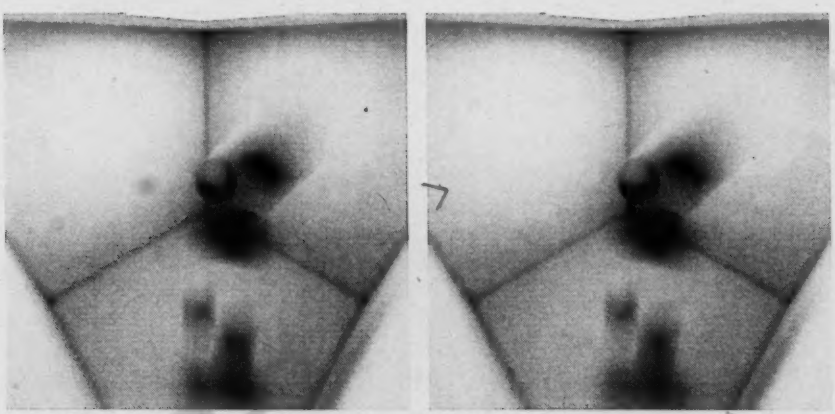

Figure 3.6: Scene containing nine spheres from the moving spheres test suite. Produced using precalculated irradiance (left) and on demand caching (right) with a time dependent photon map.

The test suite consists of ten scenes, each of which contains a different number of moving spheres. Each scene in the suite is of an enclosed $40 \times 40 \times 40$ rectangular box with a 10×10 rectangular area light (with $I_{\mathcal{L}} =< 1, 1, 1 >$ and $W_{\mathcal{L}} = 100$) at the top, and contains an odd number of diffuse moving spheres. The spheres in each scene are generated such that no two spheres intersect at any point along their trajectory, nor will any sphere intersect a wall of the box. Furthermore, the spheres are generated such that the scene with $2(i+1)+1$ spheres contains the same $2i+1$ spheres as the scene with $2i+1$ spheres. Lastly, each sphere has a radius in the range $[3, 5]$, and each component of the velocity of a sphere is generated in the range $[-5, 5]$.

Each scene in the test suite was ray traced as a 400×400 pixel image using the classical ray tracing algorithm with $N_s = 20$, $N_t = 10$, $n_p = 1500$, and only the photon map was used to calculate irradiance. Figure 3.6 shows the scene with nine moving spheres that was produced as part of this test. For further examples see Appendix D. The classical ray tracing algorithm was chosen to ray trace the scenes so that the rendered images would be a visualization of the photon map. The computer used to ray trace the scenes was a 2.26 GHz Pentium 4 with 512Mb of physical memory running Red Hat Linux (kernel version

2.4.9-31) and gcc version 3.3.

Table 3.3 shows the number of time dependent photons used in the photon map of each scene in the test suite.

Number of Moving Spheres	Number of Photons
1	1089582
3	1287577
5	1827044
7	2073026
9	2267943
11	2510410
13	2706363
15	2853756
17	2853515
19	2948030

Table 3.3: Number of time dependent photons used in each of the odd numbered moving spheres scenes.

Figure 3.7 is a comparison of the time taken to ray trace the scenes in the test suite using both Christensen's irradiance precalculation optimization and the on demand irradiance caching presented in this thesis. The time taken to ray trace each image in the test suite using the on demand irradiance caching method presented in this thesis is a fraction of the time taken to ray trace each image using Christensen's irradiance precalculating method. At one end of the graph is the scene containing one moving sphere. This scene takes 2495 seconds to ray trace using Christensen's method, but takes 509 seconds to ray trace using the on demand method presented in this thesis; only 20.4% of the time taken by Christensen's method. At the other end of the graph is the scene containing 19 moving spheres. This scene takes 13,003 seconds to ray trace using Christensen's method, but takes only 3207 seconds to ray trace using the the on demand method; 24.7% of the time taken by Christensen's method.

To compare the image quality of Christensen's irradiance precalculation method and the on demand irradiance caching method, high quality renderings of the scenes containing between one and nine moving spheres, inclusive, were ray traced using the adaptive photon map described in Section 3.3 with the on demand irradiance caching method. More high quality images were not generated due to memory constraints of the computer being used to generate them. The high quality images were ray traced using the classical ray tracing algorithm with $N_s = 20$, $N_t = 10$, $N_p = 10,000,000$, $N_{tp} = 5$, $n_p = 8,000$, and only used the photon map for illumination. The adaptive photon map was used to generate these high quality images, rather than the time dependent photon map, to reduce the time taken and memory required to generate the images. Each of these high quality images was then compared to the lower quality images by taking the Euclidean distance between pixel colours, and calculating the mean, median, standard deviation, and variance of the distances

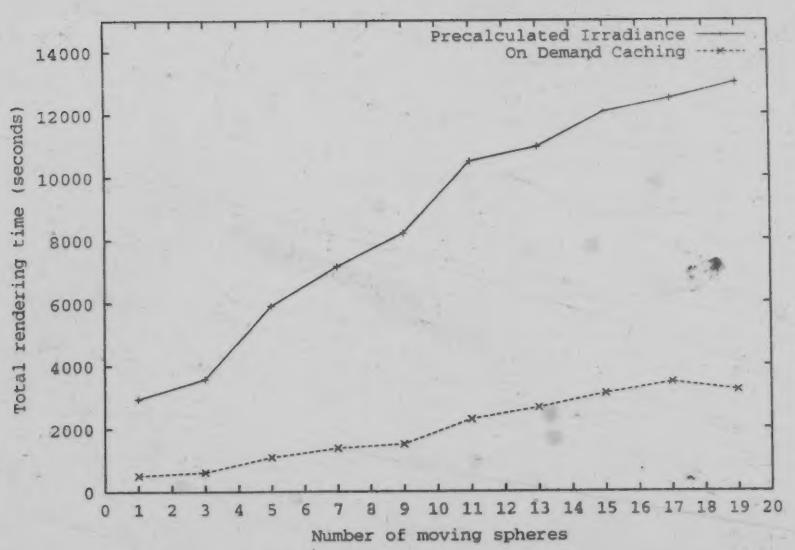

Figure 3.7: Time (in seconds) to ray trace the odd numbered moving spheres scenes in the test suite with the time dependent photon map with both Christensen's irradiance precalculation optimization and the presented on demand caching optimization.

over the whole image.

Number of Moving Spheres	Mean	Median	Standard Deviation	Variance
1	0.032418	0.030122	0.023477	0.000551
3	0.027895	0.022866	0.029272	0.000857
5	0.027543	0.015686	0.046833	0.002193
7	0.031912	0.014673	0.058643	0.003439
9	0.032126	0.014673	0.056614	0.003205

Table 3.4: Comparison of pixel colours between images rendered using the time dependent photon map with Christensen's irradiance precalculation method and the high quality baseline renders.

Number of Moving Spheres	Mean	Median	Standard Deviation	Variance
1	0.032184	0.029607	0.023261	0.000541
3	0.027721	0.022866	0.029286	0.000858
5	0.027423	0.014673	0.046790	0.002189
7	0.031849	0.014673	0.058641	0.003439
9	0.032166	0.014673	0.056687	0.003213

Table 3.5: Comparison of pixel colours between images rendered using the time dependent photon map with the on demand irradiance caching method and the high quality baseline renders.

Tables 3.4 and 3.5 show the mean, median, standard deviation, and variance of the Euclidean distance between pixel colours of the high quality images to the images generated

with Christensen's irradiance precalculation method and the on demand irradiance caching method, respectively. The mean, median, standard deviation, and variance of the two different methods are virtually identical; in fact, a number of values are identical. The difference between the two is likely due to variance in the rays used to ray trace the images using the two methods.

3.3 The Adaptive Photon Map

The time dependent photon mapping algorithm presented by Cammarano and Jensen has fixed space requirements that are solely a function of the user defined parameter N_p; the time dependent photon map requires the same amount of physical memory regardless of the number, and size, of moving objects in the scene. In this section a new adaptive photon map that merges time independent, and time dependent photons into a single photon map is presented. This adaptive photon map merges the two types of photons in a manner that adapts the number of time dependent photons to the characteristics of motion within the scene. Both passes of the standard photon mapping algorithm are modified to utilize the two photon maps.

The photon tracing pass is modified in much the same manner as adding adaptive motion blurring to a ray tracer. The photon trace will generate N_p random photons, some of which may be split into N_{tp} temporal photons.

When generating photons from a stationary light source, $\mathcal{L}$, $N_{\mathcal{L}}$ random photons are generated in exactly the same manner as in the standard, time independent, photon map. The energy of each time independent photon will be $\Phi_p = \frac{\Phi_{\mathcal{L}} \times t_s}{N_{\mathcal{L}}}$ where $\Phi_{\mathcal{L}}$ is the energy of the light source, t_s is the shutter speed, and $N_{\mathcal{L}}$ is the number of photons cast from this light source. Each time independent photon is traced through the scene by finding the nearest intersection with a stationary object, then testing if the photon passes through the path of a moving object prior to reaching the stationary object. If the photon does not pass through the path of a moving object, then it is reflected, or stored, as in the standard photon mapping algorithm. On the other hand, if the photon passes through the path of a moving object the photon is split into N_{tp} time dependent photons, each at a random time within the exposure interval. To ensure that the time dependent photons are spread out over the exposure interval, a jittered regular sampling of the exposure interval is used when generating the time stamp of each photon. Each of these time dependent photons has the same trajectory as the photon that was split, and energy equal to $\frac{1}{N_t}$ of the energy of the pre-split photon. Once a time independent photon has been split into N_t time dependent photons, each of the time dependent photons is traced through the scene in the manner described by Cammarano and Jensen.

On the other hand, if the light source is moving, then $N_{tp} \times N_{\mathcal{L}}$ photons are cast from

the light source in exactly the same manner as in Cammarano and Jensen's time dependent photon map. The energy of each of the photons will be $\Phi_p = \frac{\Phi_\mathcal{L} \times t_s}{N_\mathcal{L} \times N_{tp}}$.

Additionally, if the photon map being generated is a caustics photon map and the caustics-generating object is moving, then photons are generated in exactly the same manner as though the light source was moving.

The rendering pass of the photon mapping algorithm is modified to utilize both photon maps when calculating the illumination at a point in the scene. Given a time independent ray $\boldsymbol{R}$, with direction $\boldsymbol{\omega}$ intersecting the scene at $\boldsymbol{x}$, the illumination is calculated by summing the spatially nearest n_i time independent photons and n_d time dependent photons such that $n_p = n_i + n_d$:

$$I(\boldsymbol{x}, \boldsymbol{\omega}) \approx \frac{1}{\pi r^2 t_s} \left[\sum_{p=1}^{n_d} ((\boldsymbol{N} \cdot \boldsymbol{\omega}_{d,p}) \Phi_{d,p}) + \sum_{p=1}^{n_i} ((\boldsymbol{N} \cdot \boldsymbol{\omega}_{i,p}) \Phi_{i,p}) \right] \tag{3.1}$$

r is the maximum distance from $\boldsymbol{x}$ to any of the n_p photons, $\boldsymbol{\omega}_{d,p}$ and $\Phi_{d,p}$ are the incident direction and power of the p^{th} time dependent photon respectively, and $\boldsymbol{\omega}_{i,p}$ and $\Phi_{i,p}$ are the incident direction and power of p^{th} time independent photon respectively.

Given a time dependent ray, $\boldsymbol{R}$, at time t with direction $\boldsymbol{\omega}$ intersecting the scene at $\boldsymbol{x}$, the illumination is calculated by summing the spatially, and temporally, nearest n_i time independent photons and n_d time dependent photons such that $n_p = n_i + n_d$:

$$I(\boldsymbol{x}, \boldsymbol{\omega}, t) \approx \frac{1}{\pi r^2 \Delta t} \sum_{p=1}^{n_d} ((\boldsymbol{N} \cdot \boldsymbol{\omega}_{d,p}) \Phi_{d,p}) + \frac{1}{\pi r^2 t_s} \sum_{p=1}^{n_i} ((\boldsymbol{N} \cdot \boldsymbol{\omega}_{i,p}) \Phi_{i,p}) \tag{3.2}$$

where Δt is defined as in the time dependent photon map. It is important to note that since the time independent photons do not have a time associated with them, a spacial nearest neighbour search is all that is required. On the other hand, the same two-pass algorithm for finding the spatially nearest time dependent photons as used in Cammarano and Jensen's time dependent photon map is used to find the time dependent photons used.

When performing the search to find the nearest n_p photons it is quite likely that the total number of time dependent and time independent photons found exceeds n_p. In this case, each set of photons (the time dependent and time independent photons) are sorted by distance from the point $\boldsymbol{x}$. A binary search is then used to find a radius r such that the number of photons within that radius of the point $\boldsymbol{x}$ is n_p.

It should also be noted that since the photon tracing algorithm will never deposit a time independent photon on a moving object, only time dependent photons are used to calculate irradiance when $\boldsymbol{x}$ lies on a moving object.

Figure 3.8 is an example scene ray traced using direct lighting and the adaptive photon map for illumination.

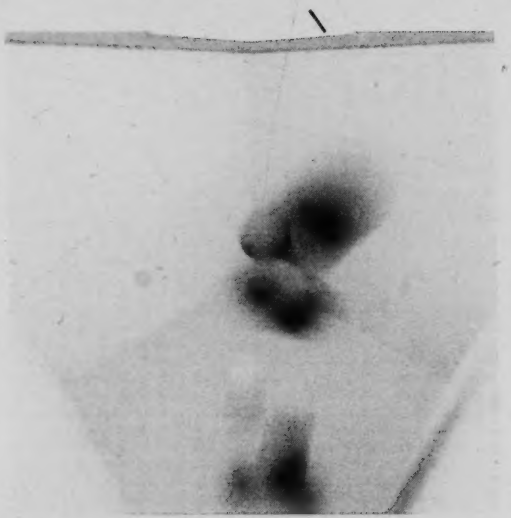

Figure 3.8: Sample image ray traced using the adaptive photon map.

3.3.1 Using On Demand Irradiance Caching

The on demand irradiance caching method presented in Section 3.2 can be used with the adaptive photon map. Time independent irradiance calculations are cached at time independent photon positions as well as time dependent photon positions. Time dependent irradiance calculations, on the other hand, only make sense when stored in a time dependent photon; time independent photons do not have a time associated with them.

3.3.2 Results

This section presents data comparing the adaptive photon map to Cammarano and Jensen's time dependent photon map. The test suite used to compare the two methods is the same test suite used in Section 3.2.1 to evaluate the on demand irradiance caching method.

The adaptive photon map for each scene was generated using $N_p = 1,000,000$ and $N_{tp} = 5$. The number of photons used in each photon map is presented in Table 3.6. Comparing this table to Table 3.3 shows that the number of photons used in the time dependent photon map is essentially the same. Figure 3.9 shows the number of time independent and time dependent photons used for each scene in the adaptive photon map. Notice that as the number of moving spheres is increased, the number of time independent photons decreases and the number of time dependent photons increases.

When ray tracing the images used in this comparison, the ray tracing algorithm used was the classical ray tracing algorithm with $N_s = 20$, $N_l = 10$, $n_p = 1800$, and only the photon map was used to calculate irradiance. Furthermore, the on demand irradiance caching method presented in Section 3.2 was used to speed up the ray tracing algorithms.

Scene Number	Number of Photons
1	1080443
3	1263372
5	1823860
7	2041147
9	2231600
11	2509476
13	2704382
15	2833055
17	2887439
19	2968825

Table 3.6: Total number of photons used in the adaptive photon map in each of the odd numbered moving spheres scenes.

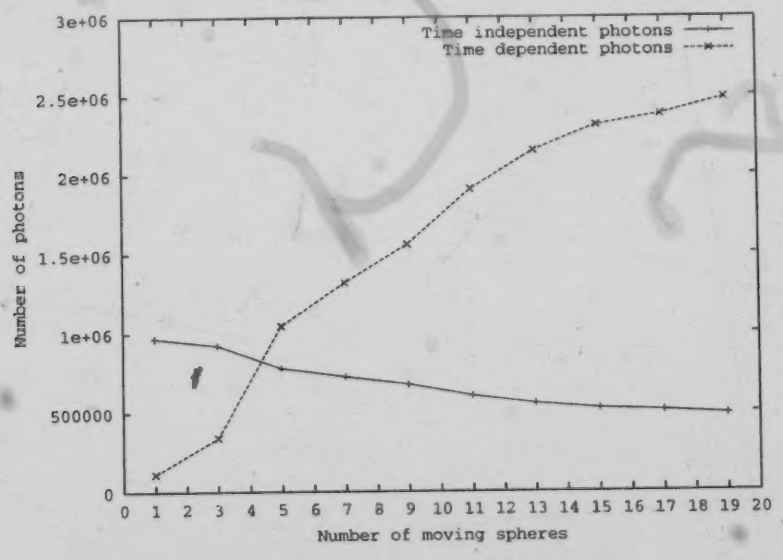

Figure 3.9: Number of time independent and time dependent photons used when ray tracing the odd numbered moving spheres scenes in the test suite.

Figure 3.10 shows a side-by-side comparison of scene five generated using a time dependent photon map and an adaptive photon map. More example images from the test suite are located in Appendix E.

Figure 3.11 shows a comparison between the amount of time taken to ray trace the scenes in the test suite using both the time dependent and the adaptive photon map. The time dependent photon map outperforms the adaptive photon map in the scenes containing relatively few moving spheres. However, the adaptive photon map outperforms the time dependent photon map for the scenes containing greater than seven moving spheres. It is not entirely clear why using the time dependent photon map yields faster times for the scenes with fewer than seven moving spheres.

Figure 3.12 shows a comparison of the amount of memory occupied by the time dependent

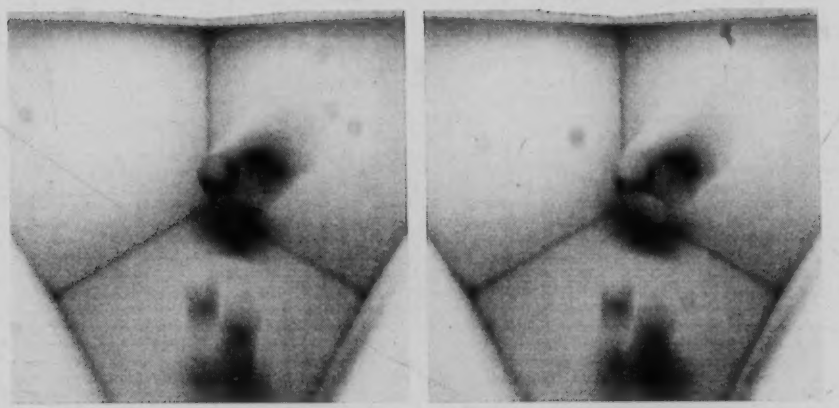

Figure 3.10: Scene containing five spheres from the moving spheres test suite. Produced using a time dependent photon map (left) and an adaptive photon map (right).

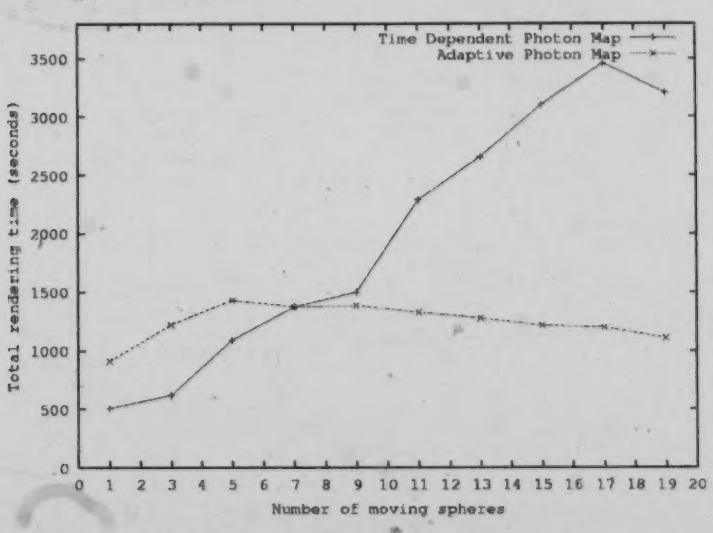

Figure 3.11: Time (in seconds) to ray trace the odd numbered moving spheres scenes in the test suite with the time dependent photon map and the adaptive photon map.

photon map and the adaptive photon map during ray tracing. The memory required by the adaptive photon map is lower than the memory required by the time dependent photon map. However, as the number of moving objects is increased the memory requirements of the adaptive photon map approach the requirements of the time dependent photon map. Reducing the resident memory usage of a photon map is important when the amount of memory used exceeds the physical memory on the computer being used. When the memory used by the photon map exceeds the physical memory on the computer, calculating the irradiance at a point can get bogged down swapping memory to and from virtual memory.

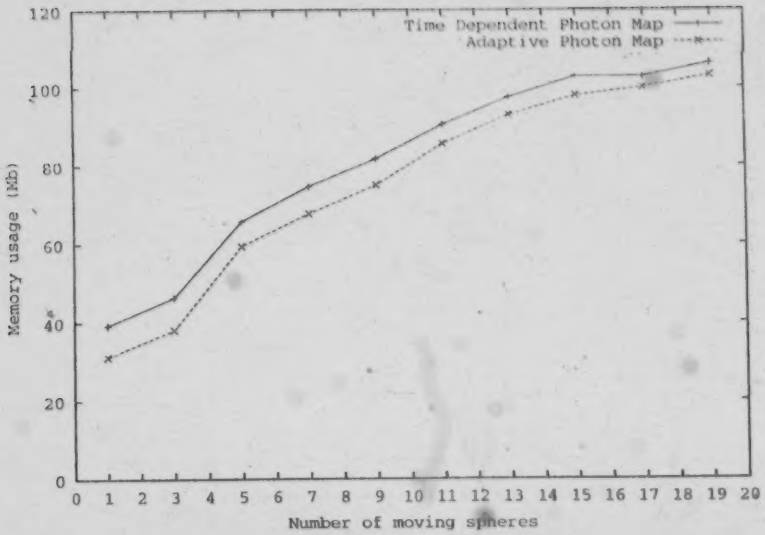

Figure 3.12: Resident memory usage (in millions of bytes) of the time dependent photon map and the adaptive photon map in the odd numbered moving spheres scenes in the test suite.

Number of Moving Spheres	Mean	Median	Standard Deviation	Variance
1	0.033701	0.033962	0.019846	0.000394
3	0.032433	0.031859	0.021564	0.000465
5	0.030898	0.028549	0.022166	0.000491
7	0.031268	0.027730	0.023903	0.000571
9	0.029920	0.025715	0.024422	0.000596

Table 3.7: Comparison of pixel colours between images rendered using the adaptive photon map with the on demand irradiance caching method and the high quality baseline renders.

Figure 3.13 shows a comparison of the peak memory usage of the adaptive photon map and the time dependent photon map. The peak memory used by a photon map is the resident memory used by the photon map plus the memory occupied by two temporary arrays created when balancing the photons into a left-balanced kd-tree. The peak memory usage of a photon map is important because if the peak usage exceeds the physical memory on the computer used, then the balancing algorithm will begin to bog down swapping memory to and from virtual memory.

To compare the difference in image quality between the adaptive photon map and the time dependent photon map, the images generated when using the adaptive photon map were compared to the same high quality images used in Section 3.2.1. Table 3.7 shows the results obtained comparing the images generated using the adaptive photon map to the high quality reference images. Comparing these values to the values in Table 3.5 shows similar mean pixel differences. The adaptive photon map produced images with a higher median

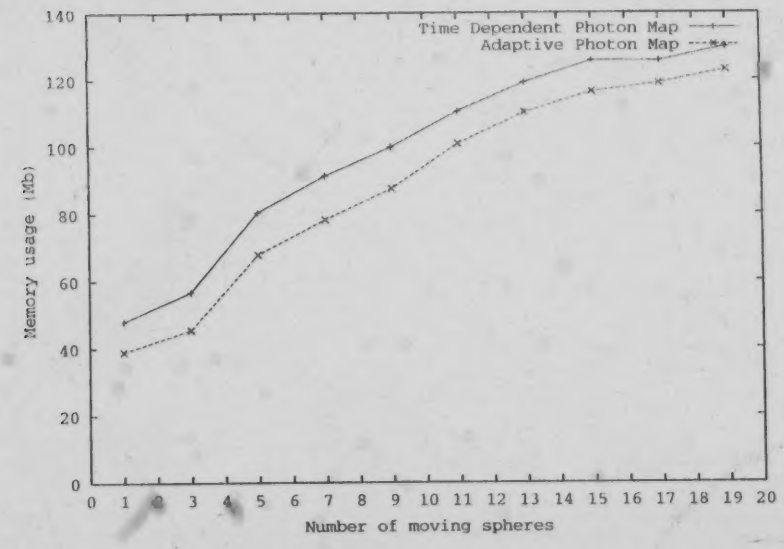

Figure 3.13: Peak memory usage (in millions of bytes) of the time dependent photon map and the adaptive photon map in the odd numbered moving spheres scenes in the test suite.

difference from the high quality images than the time dependent photon map produced. However, the standard deviation and variance of the pixel differences is substantially lower for the adaptive photon map.

When visually inspecting the images produced using both methods, some of the moving spheres produced when using the time dependent photon map appear darker than the in the image produced using the adaptive photon map. The reason for this is that when shading those regions of the image with the time dependent photon map, the photon search uses some photons which are on different spheres than the one being shaded; sometimes photons from the floors or walls are used. The result is that the area used in the irradiance calculation is much larger than it should be, and so these regions appear darker than they should. This does not occur with the adaptive photon map.

It should be noted that no scenes containing caustics were used to evaluate the adaptive photon map because of the way the adaptive photon map behaves when generating caustics. If either the light source or the caustics-generating object are moving, then the adaptive photon map will not contain any time independent photons and will be equivalent to the time dependent photon map. On the other hand, if neither the light source nor the caustics-generating object are moving, then the adaptive photon map generates caustics in exactly the same way as the time independent photon map unless there are moving objects in the path of some of the light generating the caustics. Thus, the only case that would make sense to test the adaptive photon map as a caustics map would be the case where both the light and caustics-generating objects were stationary but there is an object moving through part

of the caustics generating light. But, this case is tested just as well as a global photon map as it would as a caustics photon map.

3.4 Adaptive Photon Map with Interval Photons

The memory consumption of the adaptive photon map can further be improved upon by noting that when a time independent photon is split into N_{tp} time dependent photons, not all of the N_{tp} photons created will necessarily strike a moving object. This observation can be exploited by adding a third photon map, consisting of interval photons, to the adaptive photon map. An interval photon represents the illumination from a light source over a contiguous interval of time. Technically, the standard time independent photon can be thought of as an interval photon where the interval is equal to the exposure time of the camera.

Each interval photon is represented as a 28-byte structure:

```
struct IntervalPhoton {
  // Photon position
  float x, y, z;
  // photon interval: [start,end]
  float start, end;
  struct {
    // kd-tree flag
    unsigned short flag:2;
    // unused space
    unsigned short unused:14;
  };
  // Photon energy; compressed
  unsigned char pow[4];
  // Incident direction
  unsigned char theta, phi;
};
```

An interval photon stores the same information as a time independent photon (position, energy, incident direction, and a flag) as well as the start and end time of the photon's existence. That is, an interval photon exists at its stored location at every time t such that $t \in [\text{start}, \text{end}]$. It is important to note that the memory savings gained by using interval photons is because each interval photon represents a minimum of two time dependent photons; thus, 28 bytes of memory is used in place of at least 48 bytes of memory.

The photon tracing pass of the algorithm differs from the adaptive photon mapping algorithm only when a time independent photon has to be split into time dependent photons. Instead of splitting the time independent photon into N_{tp} time dependent photons, N_{tp} temporal samples are created; as in our adaptive photon mapping algorithm, these temporal samples are generated by jittering a regular sampling of the exposure interval. Each temporal sample, $t_1 < t_2 < t_3 < \ldots < t_{N_t}$, is then placed into a set, S, of tuples:

$$S = \{(t_i, i) : i = 1 \ldots N_t, \forall j < i, t_j < t_i\}.$$

This set, S, is used in the remainder of the photon trace. For each temporal sample $(t, i) \in S$, determine if the photon has intersected a moving object at time t. If it has, then a time dependent photon is created at time t with energy $\Phi = \frac{\Phi_{\mathcal{L}} \times t_s}{N_{\mathcal{L}} \times N_t}$ where $\Phi_{\mathcal{L}}$ is the energy of the light source from which this photon originated, t_s is the shutter speed, and $N_{\mathcal{L}}$ is the number of photons cast from the source light. The energy of the photon is decreased by Φ to reflect the energy lost to the time dependent photon that was created. This time dependent photon is then traced as in the adaptive photon map, and in Cammarano and Jensen's algorithm. Once a photon has been tested for intersection with a moving object at each of the samples in S, the photon is traced in the same manner we would trace a time independent photon with the exception that the test to split the photon is no longer performed (it has already been split).

When a photon with energy Φ and a non-empty set, S, needs to be stored into the photon map the elements of S are partitioned into non-empty subsets, $S_1, S_2, \ldots$, such that:

$$(t_i, i) \in S_j \Rightarrow \forall k \neq j : \{(t_{i-1}, i-1), (t_{i+1}, i+1)\} \cap S_k = \emptyset.$$

A photon is then stored for each partition, S_j, the type of which depends on the number of elements in S_j. If $|S_j| = 1$, then $S_j = \{(t, i)\}$ and so a time dependent photon is stored at time t with energy $\frac{\Phi}{|S|}$. On the other hand, if $|S_j| > 1$ an interval photon with start and end times equal to the minimum and maximum time in S_j, respectively, and energy $\frac{\Phi |S_j|}{|S|}$ is stored.

The rendering pass of the algorithm is modified to utilize these new interval photons as well. Given a time independent ray, $\boldsymbol{R}$, with direction $\boldsymbol{\omega}$ intersecting the scene at $\boldsymbol{x}$ the illumination at $\boldsymbol{x}$ is calculated by summing the spatially nearest n_i time independent photons, n_d time independent photons, and n_n interval photons such that $n_p = n_i + n_n + n_d$:

$$I(\boldsymbol{x}, \boldsymbol{\omega}) \approx \frac{1}{\pi r^2 t_s} \sum_{p=1}^{n_d} ((\boldsymbol{N} \cdot \boldsymbol{\omega}_{d,p}) \Phi_{d,p}) + \frac{1}{\pi r^2 t_s} \left[\sum_{p=1}^{n_n} ((\boldsymbol{N} \cdot \boldsymbol{\omega}_{n,p}) \Phi_{n,p}) + \sum_{p=1}^{n_i} ((\boldsymbol{N} \cdot \boldsymbol{\omega}_{i,p}) \Phi_{i,p}) \right], \tag{3.3}$$

where r, $\boldsymbol{\omega}_{d,p}$, $\Phi_{d,p}$, $\boldsymbol{\omega}_{i,p}$, and $\Phi_{i,p}$ are as in Equation 3.1. Also, $\boldsymbol{\omega}_{n,p}$ and $\Phi_{n,p}$ are the incident direction and power of the p^{th} interval photon, respectively.

Given a time dependent ray, $\boldsymbol{R}$, at time t with direction $\boldsymbol{\omega}$ intersecting the scene at $\boldsymbol{x}$, the illumination is calculated by summing the spatially, and temporally, nearest n_i time independent photons and n_d time dependent photons. We also include the n_n spatially nearest interval photons that have t in their interval of existence, such that $n_p = n_i + n_n + n_d$:

$$I(\boldsymbol{x},\boldsymbol{\omega},t) \approx \frac{1}{\pi r^2 \Delta t}\sum_{p=1}^{n_d}((\boldsymbol{N}\cdot\boldsymbol{\omega}_{d,p})\Phi_{d,p}) + \frac{1}{\pi r^2 t_s}\left[\sum_{p=1}^{n_n}((\boldsymbol{N}\cdot\boldsymbol{\omega}_{n,p})\Phi_{n,p}) + \sum_{p=1}^{n_i}((\boldsymbol{N}\cdot\boldsymbol{\omega}_{i,p})\Phi_{i,p})\right]. \tag{3.4}$$

Note that the photon tracing algorithm will never deposit time independent nor interval photons on a moving object. Thus, only time dependent photons are used to calculate the irradiance when $\boldsymbol{x}$ lies on a moving object.

Figure 3.14: Sample image ray traced using the adaptive photon map with interval photons.

Figure 3.14 is an example scene ray traced using direct lighting and the adaptive photon map with interval photons for illumination.

3.4.1 Using On Demand Irradiance Caching

The on demand irradiance caching method presented in Section 3.2 can be used with the adaptive photon map with interval photons in exactly the same way as in the adaptive photon map without interval photons. Although it is possible to cache time independent irradiance calculations in the interval photons, they are not used to cache irradiance calculations at all.

3.4.2 Results

This section presents data comparing the three photon maps that support time dependent illumination: the time dependent photon map, the adaptive photon map, and the time dependent photon map with interval photons. The three photon maps are compared using the same methods used to compare the adaptive photon map to the time dependent photon map.

The adaptive photon map with interval photons for each scene was generated using the same parameters as the adaptive photon map: $N_p = 1,000,000$ and $N_{tp} = 5$. The number of photons used in each photon map is presented in Table 3.8. Comparing to Table 3.6 and Table 3.3 shows that the number of photons in the the adaptive photon map with interval photons is less than the number of photons used in the other two photon maps. Figure 3.16 shows the number of each of the three types of photons used in the adaptive photon map with interval photons for each of the scenes in the test suite.

Scene Number	Number of Photons
1	1045658
3	1154007
5	1559953
7	1703090
9	1849994
11	2087074
13	2230509
15	2362293
17	2423871
19	2526054

Table 3.8: Total number of photons used in the adaptive photon map with interval photons in each of the odd numbered moving spheres scenes.

Figure 3.15 shows a side-by-side comparison of the scene containing five moving spheres generated using an adaptive photon map, and an adaptive photon map with interval photons. For more example images from the test suite see Appendix E.

Figure 3.17 is a comparison of the time taken to ray trace the scenes in the test suite using all three photon maps. For the scenes with three or fewer moving spheres, adding interval photons to the adaptive photon map grants a slight speedup in the time taken to ray trace the images. However, adding interval photons to the adaptive photon map only serves to slow down the ray tracer when more than five moving spheres are in the scene. The adaptive photon map with interval photons being outperformed by the adaptive photon map is expected because the number of time dependent photons in the photon map is reduced. Reducing the number of time dependent photons by using interval photons reduces the density of photons on stationary surfaces; thus, a larger search radius will likely be required to find the photons for an irradiance estimate on a stationary surface. On the other hand,

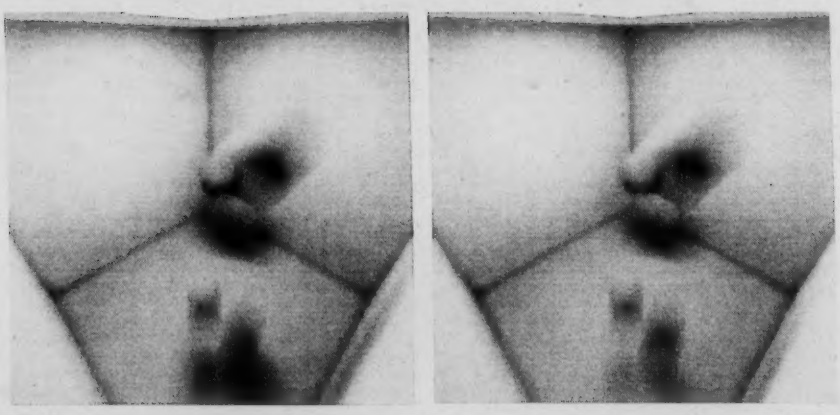

Figure 3.15: Scene containing five spheres from the moving spheres test suite. Produced using an adaptivee photon map (left) and an adaptive photon map with interval photons (right).

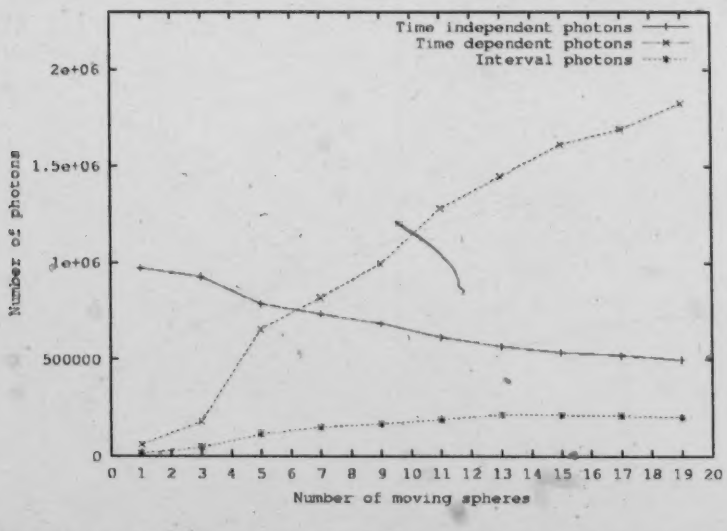

Figure 3.16: Number of interval, time independent, and time dependent photons used when ray tracing the odd numbered moving spheres scenes in the test suite.

the adaptive photon map with interval photons outperforming the adaptive photon map in the first few scenes was not expected. One possibility is that the speedup is due to the overall reduced number of photons in the photon map; having sufficiently fewer photons in a photon map will speed up searches through the photon map, even though more searches might be required to calculate an irradiance estimate. Another possibility is that the added performance of the adaptive photon map with interval photons is due to the on demand irradiance cache.

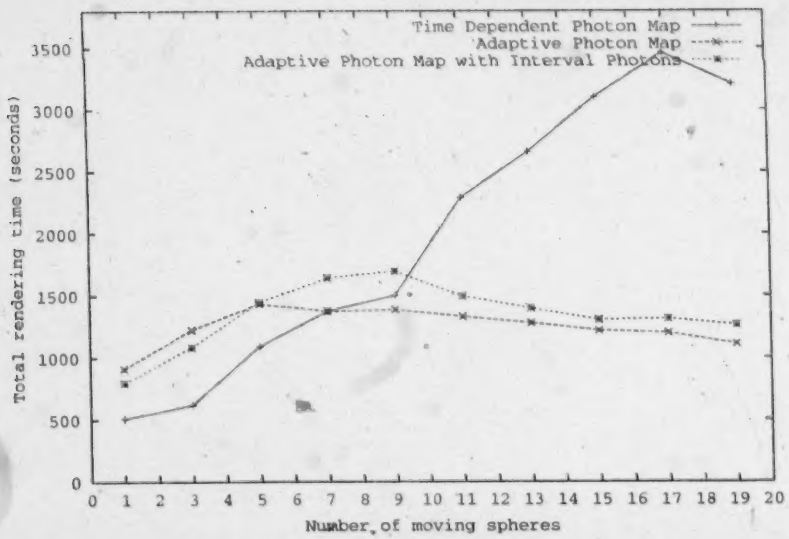

Figure 3.17: Time (in seconds) to ray trace the odd numbered moving spheres scenes in the test suite with the time dependent photon map, the adaptive photon map, and the adaptive photon map with interval photons.

Figure 3.18 shows a comparison of the resident memory footprint of all three photon maps. For scenes containing only a handful of moving spheres, adding interval photons to the adaptive photon map marginally reduces the amount of memory required by the photon map. However, as the number of moving spheres is increased the memory savings become more drastic; requiring only 80.4% of the memory required by the time dependent photon map for the scene containing 19 moving spheres.

Figure 3.19 shows a comparison of the peak memory usage of the three photon maps. Adding interval photons to the adaptive photon map reduces the peak memory usage of the photon map even further. As the number of moving spheres is increased, the memory savings grow; with the adaptive photon map with interval photons requiring over 20Mb less space than the adaptive photon map without interval photons.

Number of Moving Spheres	Mean	Median	Standard Deviation	Variance
1	0.037431	0.034412	0.030216	0.000913
3	0.038826	0.033962	0.036226	0.001312
5	0.039910	0.030628	0.042802	0.001832
7	0.040489	0.029607	0.044099	0.001945
9	0.041211	0.027730	0.047798	0.002285

Table 3.9: Comparison of pixel colours between images rendered using the adaptive photon map with interval photons plus on demand irradiance caching method and the high quality baseline renders.

Table 3.9 shows the results obtained by comparing the images generated using the adap-

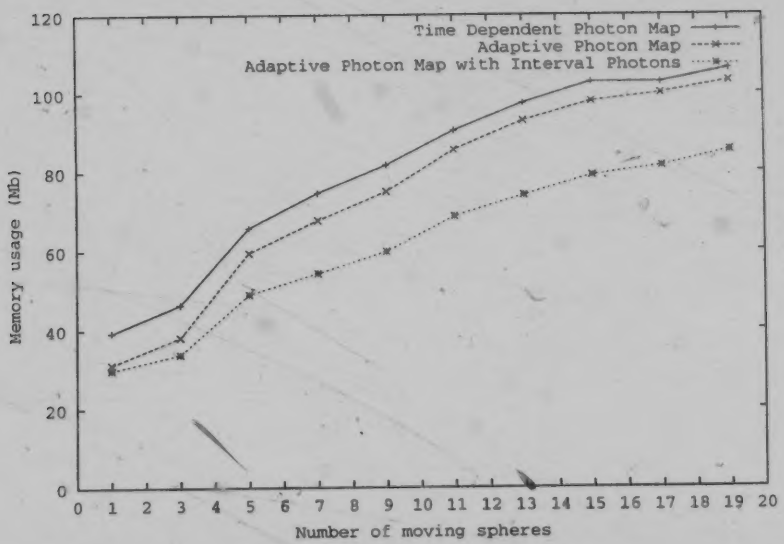

Figure 3.18: Resident memory usage (in millions of bytes) of the time dependent photon map, the adaptive photon map, and the adaptive photon map with interval photons in the odd numbered moving spheres scenes in the test suite.

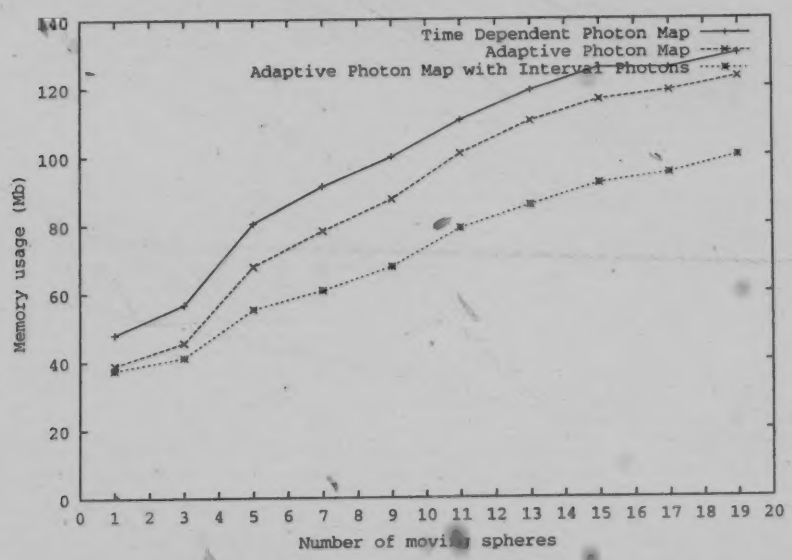

Figure 3.19: Peak memory usage (in millions of bytes) of the time dependent photon map, the adaptive photon map, and the adaptive photon map with interval photons in the odd numbered moving spheres scenes in the test suite.

tive photon map with interval photons to the high quality reference images. Comparing these values to the values in Table 3.5 shows that the mean and median values are greater when adding interval photons than in the time dependent photon map, but the variance and standard deviation are lower in the adaptive photon map with interval photons than in the

time dependent photon map. Comparing the values in Table 3.9 to Table 3.7 shows that adding interval photons to the adaptive photon map increases the median, mean, standard deviation, and the variance.

Chapter 4

Conclusion

4.1 Contributions

This thesis presented four improvements to existing ray tracing algorithms that improve the image quality, ray tracing speed, and the memory requirements of the improved algorithms.

An improvement was made to the adaptive motion blurring algorithm by combining the temporal samples that miss a moving object into a single weighted sample. This simple modification improves image quality when a small number of rays are used to calculate the colour of each pixel, and reduces the time required to ray trace scenes with motion blur.

Christensen's irradiance precalculation algorithm for photon maps was improved upon by caching irradiance calculation as they are needed rather than blindly precalculating irradiance at locations that may never be used. This improvement reduced the time required to ray trace ten test scenes to 20 to 25 percent of the time required to ray trace the same scenes using Christensen's method while not increasing the memory requirements nor degrading the quality of the ray trace compared to Christensen's method.

An adaptive photon map that merges a time independent and time dependent photon map into a single photon map was presented. This new photon map adapts the number of time dependent photons used in the photon map to the characteristics of motion in the scene being ray traced. As an improvement to Cammarano and Jensen's time dependent photon map, this new adaptive photon map reduces the amount of memory required by the time dependent photon map and can reduce the amount of time required to generate an image using the ray tracing algorithm. Furthermore, using the adaptive photon map in place of the time dependent photon map reduces the variance of pixel values when comparing the images generated, using the two methods, to a set of high quality images generated.

Last, but not least, an improvement to the adaptive photon map was presented that adds a new type of photon, the interval photon, to the photon map. Adding this interval photon to the adaptive photon map reduces the memory required by the adaptive photon map by merging time dependent photons together into a single interval photon. Furthermore,

the adaptive photon map with interval photons is capable of outperforming both the time dependent and adaptive photon maps. However, the adaptive photon map with interval photons does not outperform the adaptive photon map in more than half of the test scenes used. As a side effect of combining two or more time dependent photons into a single interval photon the accuracy of the photon map is reduced by a small margin, when compared to a high quality version of the same image.

4.2 Future Work

When using a photon map to estimate the illumination of a scene, the parameters N_p, n_p, and N_{tp} have a large effect on the quality of the estimate and the time required to calculate an irradiance estimate. If the number of photons in the photon map (affected by N_p and N_{tp}) is too large then the photon map occupies more memory than is required, and the performance of the photon map suffers. On the other hand, if the value of n_p is too low, then the irradiance estimated by the photon map will be very blotchy. Alternately, if n_p is higher than is required to get a smooth irradiance estimate, then performance suffers. An automated algorithm for calculating appropriate values of N_p, N_{tp}, and n_p is required to make the photon mapping algorithms more accessible to the general ray tracing public.

Further investigation into scene properties that characterize which of the three photon maps, that support motion blurred illumination, would be best suited for a particular scene is also required. Right now, the only clear choice for whether to use the time dependent photon map, the adaptive photon map, or the adaptive photon map with interval photons is if the scene is being ray traced under memory constraints; if there are memory constraints, then the adaptive photon map with interval photons will require the least memory of the three methods. On the other hand, in the absence of memory constraints, it is not entirely clear which photon map will yield the best performance for a given scene. Possible avenues for investigation are the speed of the moving objects in the scene (either fast or slow), and the size of the moving objects relative to the size of the scene.

References

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics*, pages 1–10, August 1987.

[2] James Arvo. Backwards ray tracing. In *Developments in Ray Tracing*, volume 12 of *SIGGRAPH Course Notes*, 1986.

[3] Mike Cammarano and Henrik W. Jensen. Time dependent photon mapping. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 135–144, June 2002.

[4] Allen Y. Chang. A survey of geometric data structures for ray tracing. Technical Report TR-CIS-2001-06, Department of Computer and Information Science, Polytechnic University, Brooklyn, New York, 10 2001.

[5] Shudeb Chattopadhyay and Akiro Fujimoto. Bi-directional ray tracing. In *Computer Graphics*, pages 335–343. Springer-Verlag, 1987.

[6] Per H. Christensen. Faster photon map global illumination. *Journal of Graphics Tools*, 4(3):1–10, 1999.

[7] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *ACM SIGGRAPH*, pages 137–145, 1984.

[8] James Foley, Andries van Dam, Steven Feiner, and John Hughesa. *Computer Graphics: Principles and Practice.* The Systems Programming Series. Addison Wesley, second edition, November 1993.

[9] Donald Fussell and Kalpathi R. Subramanian. Fast ray tracing using k-d trees. Technical Report TR-88-07, Department of Computer Science, The University of Texas at Austin, 1988.

[10] Gaston H. Gonnet and Ricardo Baeza-Yates. *Handbook of algorithms and data structures: in Pascal and C.* Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1991.

[11] David Halliday, Robert Resnick, and Jearl Walker. *Fundamentals of Physics*, volume 4. John Wiley & Sons, Inc., fifth edition, 1997.

[12] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* PhD thesis, Czech Technical University, Prague, November 2000.

[13] Vlastimil Havran, Tomáš Kopal, Jiří Bittner, and Jiří Žára. Fast robust BSP tree traversal algorithm for ray tracing. *Journal of Graphics Tools*, 2(4):15–23, 1997.

[14] Henrik W. Jensen. Global illumination using photon maps. In *Rendering Techniques '96*, pages 21–30. Springer-Verlag, 1996.

[15] Henrik W. Jensen. *Realistic Image Synthesis Using Photon Mapping.* A.K. Peters, 2001.

[16] Francis S. Hill Jr. *Computer Graphics.* Prentice Hall, 1990.

[17] James T. Kajiya. The rendering equation. In *ACM SIGGRAPH*, pages 143–150, 1986.

[18] Krzysztof S. Klimaszwski and Thomas W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, January 1997.

[19] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[20] Michael Potmesil and Indranil Chakravarty. Modeling motion blur in computer-generated images. In *ACM SIGGRAPH*, pages 389–399, 1983.

[21] Hanan Samet and Robert E. Webber. Hierarchical data structures and algorithms for computer graphics. *IEEE Computer Graphics and Applications*, 8(4):59–75, July 1988.

[22] Tim Schroeder. Collision detection using ray casting. *Game Developer*, 8(8):50–56, August 2001.

[23] Peter Shirley. *Realistic Ray Tracing*. A.K. Peters Ltd., 2000.

[24] Brian Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998.

[25] Greg Ward. Real pixels. In *Graphics Gems*, volume II, pages 80–83. Academic Press, 1991.

[26] Gregory Ward, Francis Rubinstein, and Robert Clear. A ray tracing solution for diffuse interreflection. *Computer Graphics*, 22(4):85–92, August 1988.

[27] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques*. Addison-Wesley, 1992.

[28] Andrew Woo. Personal comminication, August 2003.

Appendix A

Glossary of Terms

- *Caustic*: A concentration of light caused by refraction or mirror reflection.
- *Caustics photon map*: A structure which records the position and properties of photons of light which form caustics.
- *Global photon map*: A structure which records the position and properties of photons of light as emitted by light sources into a scene.
- *Index of refraction*: A dimensionless quantity that helps determine the amount of refraction that takes place when light enters, or exits, a medium.
- *Motion blur*: Blurring of an object in a photograph caused by motion of the object relative to the camera.
- *Orthonormal*: A pair of vectors, $\boldsymbol{u}$ and $\boldsymbol{v}$, are orthonormal if, and only if, they are both unit length and $\boldsymbol{u} \cdot \boldsymbol{v} = 0$ (the two vectors form a 90 degree angle).
- *Photon*: A massless particle of energy. For example, a photon of light.

Appendix B

Calculating the Cohen-Sutherland Outcode of a Point

```
function calculate_outcode(point)
    outcode ← 0
    if (point.x > x_max)
        outcode = outcode bitOR CLIP_RIGHT
    else if (point.x < x_min)
        outcode = outcode bitOR CLIP_LEFT
    endif
    if (point.y > y_max)
        outcode = outcode bitOR CLIP_TOP
    else if (point.y < y_min)
        outcode = outcode bitOR CLIP_BOTTOM
    endif
    if (point.z > z_max)
        outcode = outcode bitOR CLIP_BACK
    else if (point.z < z_min)
        outcode = outcode bitOR CLIP_FRONT
    endif
    return outcode
```

where CLIP_RIGHT, CLIP_LEFT, CLIP_TOP, CLIP_BOTTOM, CLIP_BACK, and CLIP_FRONT are different powers of two.

Appendix C

Images Using Adaptive Motion Blurring

This appendix contains a sampling of the images generated to compare the image quality of the adaptive motion blurring algorithm to the VIAMB algorithm. The comparison was done in Section 3.1.2.

Figure C.1: Scene one from the test suite used to compare the quality of the two adaptive motion blurring algorithms. The left image was produced using the original algorithm, and the right was produced using VIAMB.

Figure C.2: Scene 14 from the test suite used to compare the quality of the two adaptive motion blurring algorithms. The left image was produced using the original algorithm, and the right was produced using VIAMB.

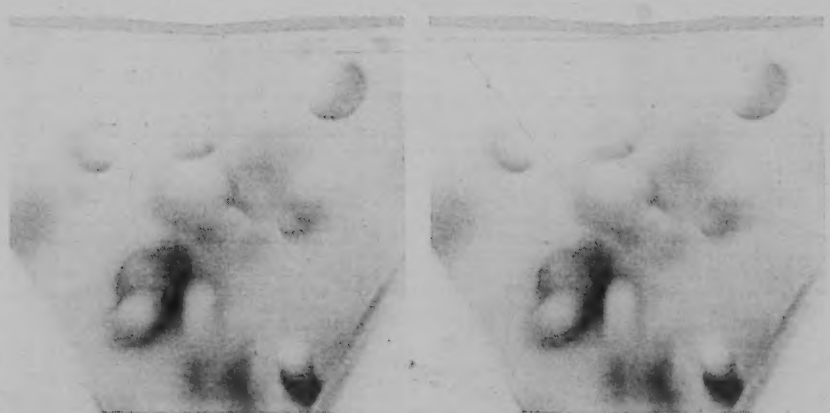

Figure C.3: Scene 18 from the test suite used to compare the quality of the two adaptive motion blurring algorithms. The left image was produced using the original algorithm, and the right was produced using VIAMB.

Appendix D

Images Using Precalculated Irradiance and On Demand Caching

This chapter contains a sampling of the images used to compare Christensen's irradiance precalculation method to the on demand irradiance caching method. The comparison was done in Section 3.2.1.

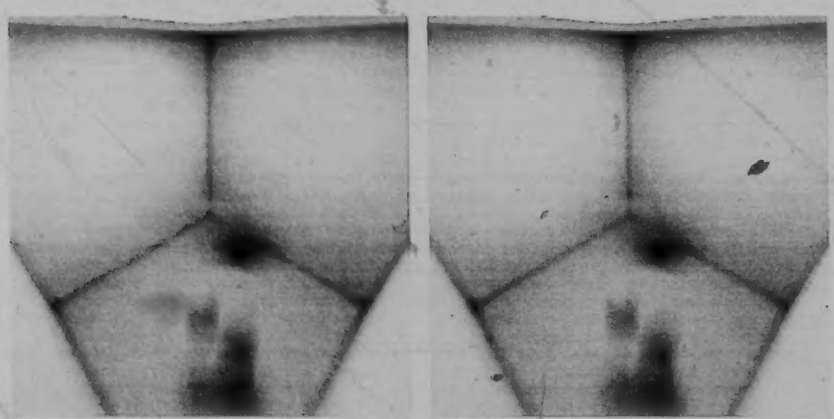

Figure D.1: Scene containing three spheres from the moving spheres test suite. Produced using precalculated irradiance (left) and on demand caching (right) with a time dependent photon map.

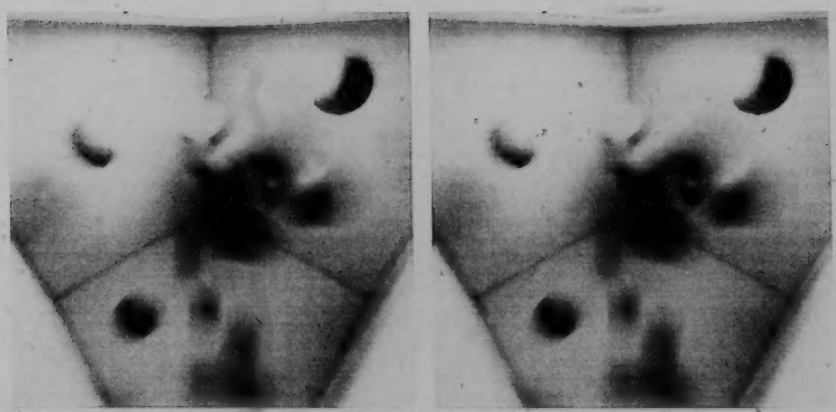

Figure D.2: Scene containing 13 spheres from the moving spheres test suite. Produced using precalculated irradiance (left) and on demand caching (right) with a time dependent photon map.

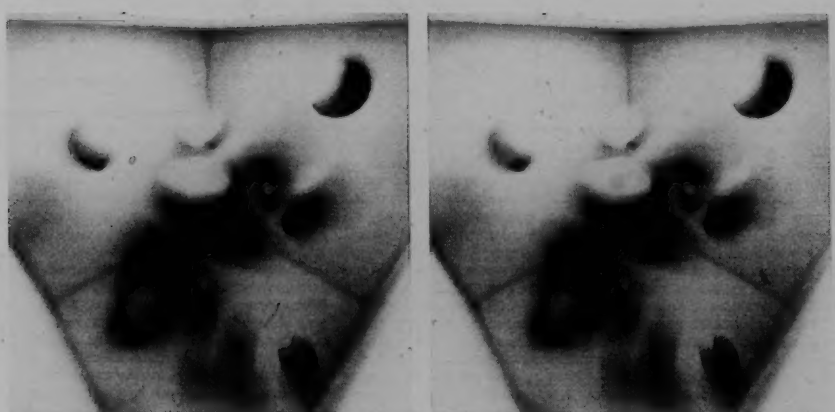

Figure D.3: Scene containing 19 spheres from the moving spheres test suite. Produced using precalculated irradiance (left) and on demand caching (right) with a time dependent photon map.

Appendix E

Images Using the Time Dependent Photon Maps

This appendix contains a sampling of the images generated to evaluate the effectiveness of the adaptive photon map, with and without interval photons, as compared to Cammarano and Jensen's time dependent photon map.

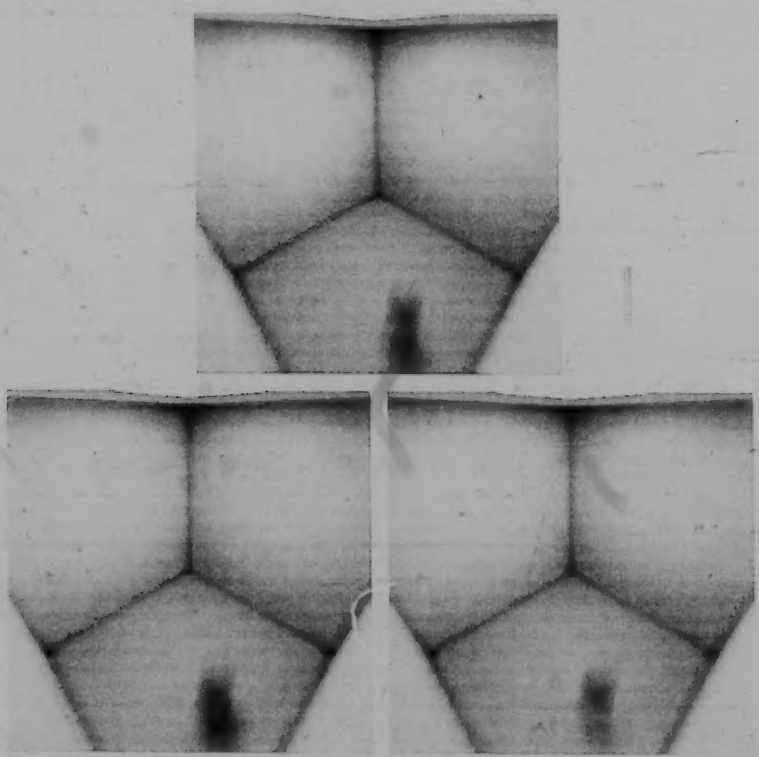

Figure E.1: Scene containing one sphere from the moving spheres test suite. Produced using the time dependent photon map (top), the adaptive photon map (bottom-left), and the adaptive photon map with interval photons (bottom-right).

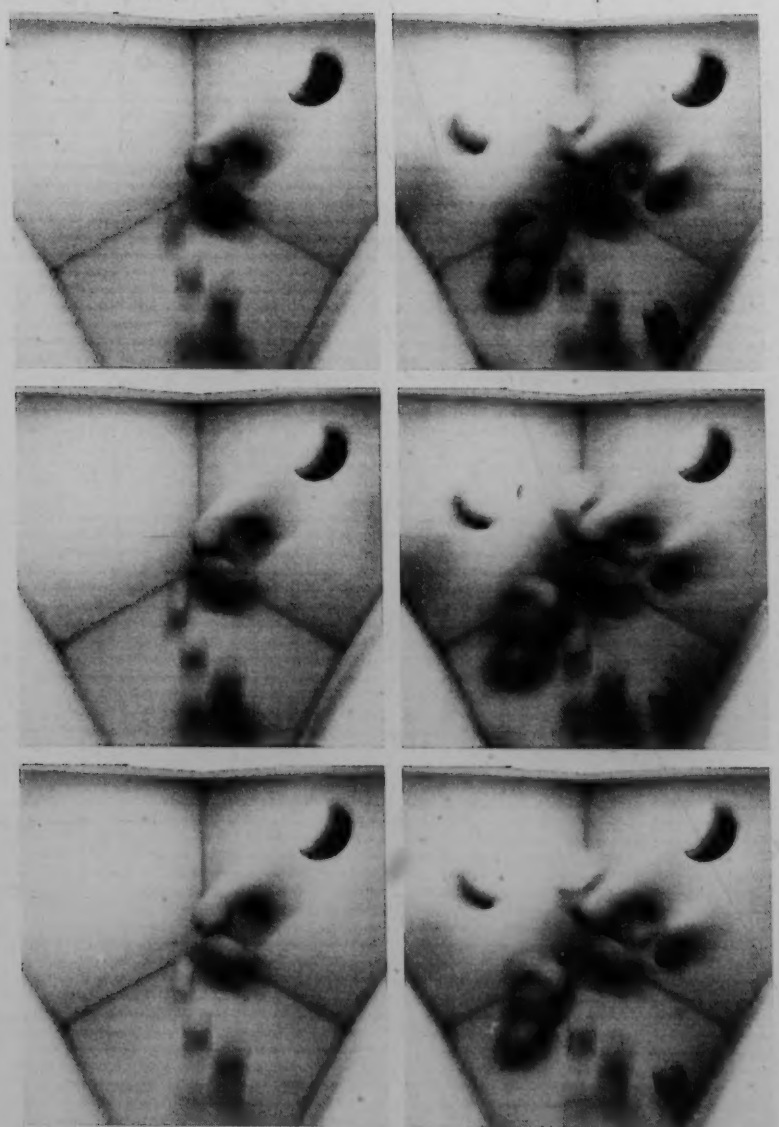

Figure E.2: Scenes containing seven (left) and 17 (right) spheres from the moving spheres test suite. Produced using the time dependent photon map (top), the adaptive photon map (middle), and the adaptive photon map with interval photons (bottom).

Appendix F

Specifications of the Computers Used to Generate Data

This appendix contains a record of the specs of the three identical computers used to generate the data for this thesis.

F.1 giroux.cs.ualberta.ca

F.1.1 uname -a

```
Linux giroux 2.4.9-31uofa #1 Fri May 17 02:34:33 MDT 2002 i686 unknown
```

F.1.2 cat /proc/version

```
Linux version 2.4.9-31uofa (root@goodfare) (gcc version 2.96 20000731 (Red
Hat Linux 7.1 2.96-98)) #1 Fri May 17 02:34:33 MDT 2002
```

F.1.3 cat /proc/cpuinfo

```
processor : 0
vendor_id : GenuineIntel
cpu family : 15
model : 2
model name : Intel(R) Pentium(R) 4 CPU 2.26GHz
stepping : 4
cpu MHz : 2254.028
cache size : 512 KB
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 2
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
        pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
bogomips : 4495.76
```

F.1.4 cat /proc/meminfo

```
        total:    used:    free:  shared: buffers:  cached:
Mem:  524992512 390094848 134897664     4096 96960512 266403840
Swap: 2097434624 149798912 1947635712
```

```
MemTotal:       512688 kB
MemFree:        131736 kB
MemShared:           4 kB
Buffers:         94688 kB
Cached:         119604 kB
SwapCached:     140556 kB
Active:         196008 kB
Inact_dirty:    158844 kB
Inact_clean:         0 kB
Inact_target:   130932 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       512688 kB
LowFree:        131736 kB
SwapTotal:     2048276 kB
SwapFree:      1901988 kB
```

F.1.5 gcc -v

```
Reading specs from
   /usr/anzac/local/gcc-3.3/lib/gcc-lib/i686-pc-linux-gnu/3.3/specs
Configured with: /usr/local/depot/gcc/gcc-3.3/src/orig/configure
                 --prefix=/usr/anzac/local/gcc-3.3
Thread model: posix
gcc version 3.3
```

F.2 garner.cs.ualberta.ca

F.2.1 uname -a

```
Linux garner 2.4.17 #2 Wed Aug 7 03:06:35 MDT 2002 i686 unknown
```

F.2.2 cat /proc/version

```
Linux version 2.4.17 (root@garner) (gcc version 2.96 20000731 (Red
Hat Linux 7.1 2.96-98)) #2 Wed Aug 7 03:06:35 MDT 2002
```

F.2.3 cat /proc/cpuinfo

```
processor : 0
vendor_id : GenuineIntel
cpu family : 15
model : 2
model name : Intel(R) Pentium(R) 4 CPU 2.26GHz
stepping : 4
cpu MHz : 2254.026
cache size : 512 KB
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 2
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
        pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
bogomips : 4495.76
```

F.2.4 cat /proc/meminfo

```
        total:    used:    free:  shared: buffers:  cached:
Mem:  525676544 375246848 150429696        0 88915968 105267200
Swap: 2097434624  8351744 2089082880
MemTotal:       513356 kB
MemFree:        146904 kB
MemShared:           0 kB
Buffers:         86832 kB
Cached:         101036 kB
SwapCached:       1764 kB
Active:         106944 kB
Inactive:        86540 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       513356 kB
LowFree:        146904 kB
SwapTotal:     2048276 kB
SwapFree:      2040120 kB
```

F.2.5 gcc -v

```
Reading specs from
  /usr/anzac/local/gcc-3.3/lib/gcc-lib/i686-pc-linux-gnu/3.3/specs
Configured with: /usr/local/depot/gcc/gcc-3.3/src/orig/configure
    --prefix=/usr/anzac/local/gcc-3.3
Thread model: posix
gcc version 3.3
```

F.3 goodfare.cs.ualberta.ca

F.3.1 uname -a

```
Linux goodfare 2.4.9-31uofa #1 Fri May 17 02:34:33 MDT 2002 i686 unknown
```

F.3.2 cat /proc/version

```
Linux version 2.4.9-31uofa (root@goodfare) (gcc version 2.96 20000731 (Red
Hat Linux 7.1 2.96-98)) #1 Fri May 17 02:34:33 MDT 2002
```

F.3.3 cat /proc/cpuinfo

```
processor : 0
vendor_id : GenuineIntel
cpu family : 15
model : 2
model name : Intel(R) Pentium(R) 4 CPU 2.26GHz
stepping : 4
cpu MHz : 2254.023
cache size : 512 KB
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 2
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
        pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
bogomips : 4495.76
```

F.3.4 cat /proc/meminfo

```
        total:    used:    free:  shared: buffers:  cached:
Mem:  524992512 417316864 107675648     8192 105455616 274755584
Swap: 2097434624 150794240 1946640384
MemTotal:       512688 kB
MemFree:        105152 kB
MemShared:           8 kB
Buffers:        102984 kB
Cached:         124416 kB
SwapCached:     143900 kB
Active:         242960 kB
Inact_dirty:    128156 kB
Inact_clean:       192 kB
Inact_target:   130932 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       512688 kB
LowFree:        105152 kB
SwapTotal:     2048276 kB
SwapFree:      1901016 kB
```

F.3.5 gcc -v

```
Reading specs from
  /usr/anzac/local/gcc-3.3/lib/gcc-lib/i686-pc-linux-gnu/3.3/specs
Configured with: /usr/local/depot/gcc/gcc-3.3/src/orig/configure
  --prefix=/usr/anzac/local/gcc-3.3
Thread model: posix
gcc version 3.3
```